

Notes on Modern C++

Gyubeom Edward Im*

December 13, 2024

Contents

1	Introduction	3
2	Intermediate	13
2.1	Temporary	13
2.1.1	return-by-value vs return-by-reference	13
2.1.2	temporary and casting	14
2.2	Conversion	14
2.2.1	Conversion constructor, conversion operator	15
2.2.2	explicit constructor	16
2.2.3	explicit conversion operator	17
2.2.4	explicit(bool), c++20	18
2.2.5	Conversion example: nullptr	19
2.2.6	Conversion example: return type resolver	19
2.2.7	Lambda expression and conversion	19
2.3	Constructor	20
2.3.1	Base from member idioms	21
2.3.2	Constructor and virtual function	22
2.4	This call	22
2.5	Member function pointer	24
2.5.1	std::invoke, std::mem_fn	25
2.6	Member data pointer	25
2.7	Implement custom max	26
2.7.1	Add compare operator	28
2.7.2	c++20 ranges algorithm	29
2.8	Size of member function pointer	29
2.9	new, delete and placement new	31
2.9.1	Using placement new	32
2.9.2	vector and placement new	32
2.10	Trivial constructor	33
2.10.1	Trivial default constructor	33
2.10.2	Trivial copy constructor	33
2.11	Type deduction	34
2.11.1	Auto type deduction	35
2.11.2	Array Name	36
2.12	Rvalue & forwarding & reference	37
2.12.1	Lvalue vs Rvalue	37
2.12.2	Reference & Overloading	38
2.12.3	Reference collapsing	39
2.12.4	Forwarding reference	40
2.13	Move semantics	41
2.13.1	Move constructor	41
2.13.2	std::move	42
2.13.3	Move and noexcept	42
2.13.4	Default move constructor	44
2.13.5	Rule of 3/5/0	44

*blog: alida.tistory.com, email: criterion.im@gmail.com

2.14	Perfect forwarding	45
2.14.1	Using forwarding reference	46
2.14.2	Variadic parameter template	46
2.14.3	Member function pointer for perfect forwarding	47
2.14.4	Function object for perfect forwarding	48
2.14.5	Chronometry implementation	48
2.14.6	Notice for perfect forwarding	49
2.14.7	Perfect forwarding in STL	49
2.15	Callable object	51
2.15.1	Function object	51
2.15.2	Function object = function with state	52
2.15.3	closure & function object	53
2.15.4	inline & function object	53
2.15.5	STL & function object	54
3	STL programming	55
3.1	STL structure	55
3.1.1	Generic algorithm - find	55
3.1.2	Make iterator	57
3.1.3	Generic Algorithm - list	58
3.1.4	Container, Iterator, Algorithm (CIA)	60
3.1.5	Member type	61
3.2	Iterator	62
3.2.1	Iterator concept	62
3.2.2	Container iterator	63
3.2.3	Iterator invalidation	64
3.2.4	std::ranges::begin	65
3.2.5	Iterator value type	66
3.2.6	Iterator category	67
3.2.7	Iterator operation	69
3.2.8	<code>++ vs next</code>	70
3.2.9	<code>std::copy vs std::ranges::copy</code>	71
3.2.10	Reverse iterator	72
3.2.11	Insert iterator	73
3.2.12	<code>std::counted_iterator & std::default_sentinel</code>	76
3.2.13	Iterator and sentinel	78
3.2.14	Ostream iterator	79
3.2.15	Ostreambuf iterator	79
3.2.16	Istream iterator	80
3.3	Algorithm	81
3.3.1	Erase-remove idioms	81
3.3.2	Algorithm vs member function	82
3.3.3	<code>std::erase, std::erase_if</code>	83
3.3.4	Algorithm with function as parameter	84
3.3.5	Predicate	84
3.3.6	Algorithm copy version	85
3.3.7	Projection	86
3.3.8	Constrained algorithm function object	87
4	References	89
5	Revision log	89

1 Introduction

본 포스트는 필자가 Modern c++을 공부하면서 중요하다고 생각되는 내용들을 정리한 포스트이다. 각 c++ 표준별로 개선된 사항에 대해 정리하면 다음과 같다. 대부분의 내용은 [1],[2]를 참고하여 작성하였다.

- **c++11:** 해당 버전 이전을 Legacy c++라고 하고, 이후를 Modern c++이라고 구분할 만큼 많은 변화가 있었다. rvalue reference(&&)와 이동 생성자, 이동 대입 연산자, `auto`, `lambda`, `constexpr` 등 많은 개념이 도입되었다.
 - 그외에도 `nullptr`, `char16_t`, `char32_t` 타입 추가,
 - 중괄호 초기화,
 - 멤버 선언부 초기화,
 - `range-based for`문,
 - `default`, `delete`, `override`, `final` 키워드 추가, 생성자 위임, 생성자 상속,
 - 명시적 형변환,
 - `noexcept`, `decltype` 추가,
 - perfect forwarding,
 - `static_assert()`,
 - 가변 템플릿,
 - 런타임 성능 개선과 컴파일 타임 프로그래밍, 코딩 컨벤션 강화 등 많은 부분이 추가되었다.
- **c++14:** c++11에 추가된 내용을 보강하였다. 리턴 타입 추론으로 c++11의 후행 리턴 개념을 보강하였고 `constexpr` 함수 제약을 완화하여 컴파일 타임 함수 작성 편의성을 향상하였다.
 - 그외에도 `variable template`, `decltype(auto)`, `[[deprecated]]` 키워드,
 - 람다 캡쳐 초기화, 일반화된 람다 표현식 등의 내용이 추가되었다.
- **c++17:** 기존 컴파일러에 의존하던 코드 최적화를 임시 구체화와 복사 생략 보증으로 표준화하였고 `if` `constexpr`과 클래스 템플릿 인수 추론으로 컴파일 타임 프로그래밍을 강화하였다.
 - 그외에도 `inline variable`, `auto`의 중괄호 초기화 특수 추론 규칙 개선,
 - `enum`의 중괄호 직접 초기화 허용, 람다 캡쳐시 `*this` 이용,
 - `constexpr` 람다 표현식,
 - `static_assert()`의 메세지 생략,
 - `noexcept` 함수 유형 포함,
 - Fold 표현식,
 - 비타입 템플릿 인자에서 `auto` 허용,
 - 16진수 부동 소수점 리터럴, `[[fallthrough]]`, `[[nodiscard]]`, `[[maybe_unused]]` 키워드가 추가되었다.
- **c++20:** `concept`, `requires`가 추가되어 코딩 컨벤션이 강화되었으며 모듈로 컴파일 속도가 향상되었다. 코루틴으로 함수의 일시 정지가 가능해졌으며 삼중 비교 연산자가 추가되어 비교 연산자 구현이 간단해졌다. 축약된 함수 템플릿으로 `auto`를 함수의 인자로 사용할 수 있고, 람다 표현식에서 템플릿 인자 자원으로 람다 표현식의 일반화 프로그래밍이 강화되었다. 그리고 `range-based for`문에서 초기식을 사용할 수 있어 반복문 작성이 좀 더 쉬워졌다.
 - 그외에도 컴파일 타임 프로그래밍 (`constexpr` 함수, `constinit`, `constexpr` 함수)의 추가 제약이 완화되었으며
 - 템플릿 인수 추론 시 `initializer_list`가 개선되었고
 - 람다 표현식에서 파라미터 팩 지원, 캡쳐에서 구조화된 바인딩 지원, 상태없는 람다 표현식의 기본 생성과 복사 대입 지원, 미평가 표현식에서도 람다 표현식 허용,
 - `explicit(bool)` 추가,
 - 인라인 네임스페이스와 단순한 중첩 네임스페이스 결합, 지명 초기화,
 - `new[]`에서 중괄호 집합 초기화로 배열 크기 추론,
 - `using enum` 추가, `__VA_OPT__`, `__has_cpp_attribute()` 매크로 함수 추가, `[[nodiscard]]`의 생성자 지원, `[[nodiscard("reason")]]`, `[[likely]]`, `[[unlikely]]`, `[[no_unique_address]]`가 추가되었다.

Runtime programming	
move (c++11)	(c++11) <ul style="list-style-type: none"> 우측값 참조(<code>&&</code>)와 이동 생성자, 이동 대입 연산자가 추가되어 이동 연산을 지원하며, 임시 객체 대입 시 속도가 향상되었다. 전달 참조가 추가되어 포워딩 함수에서도 효율적으로 함수 인자를 완벽하게 전달할 수 있다.
unrestricted unions (c++11)	(c++11) <ul style="list-style-type: none"> 무제한 공용체(Unrestricted unions)가 추가되어 공용체 멤버에서 생성자/소멸자/가상 함수 사용 제한이 풀렸으며, 메모리 절약을 위한 코딩 자유도가 높아졌다.
Temporary materialization & Copy elision (c++17)	(c++17) <ul style="list-style-type: none"> 임시 구체화(Temporary materialization)와 복사 생략 보증(Copy elision)을 통해 컴파일러의 존적이었던 생성자 호출 및 함수 인수 전달 최적화, 리턴값 최적화 등이 표준화되었다.

Compile time programming	
constexpr (c++11)	(c++11) <ul style="list-style-type: none"> <code>constexpr</code>이 추가되어 컴파일 타임 프로그래밍이 강화되었다. (c++14) <ul style="list-style-type: none"> <code>constexpr</code> 함수의 제약이 완화되어 지역 변수, 2개 이상 리턴문, <code>if</code>, <code>for</code>, <code>while</code> 등을 사용할 수 있게 되었다. (c++17) <ul style="list-style-type: none"> <code>if constexpr</code>이 추가되어 조건에 맞는 부분만 컴파일하고 그렇지 않은 부분은 컴파일에서 제외할 수 있다. (c++20) <ul style="list-style-type: none"> <code>constexpr</code> 함수가 추가되어 컴파일 타임 함수로만 동작할 수 있다. <code>constinit</code>이 추가되어 전역 변수, <code>static</code> 전역 변수, 정적 멤버 변수를 컴파일 타임에 초기화할 수 있다. <code>constexpr</code> 함수 제약 완화가 보강되어 가상 함수, <code>dynamic_cast</code>, <code>typeid()</code>, 초기화되지 않은 지역 변수, <code>try-catch()</code>, 공용체 멤버 변수 활성 전환, <code>asm</code> 등을 사용할 수 있다.
static_assert (c++11)	(c++11) <ul style="list-style-type: none"> <code>static_assert</code>이 추가되어 컴파일 타임 진단이 가능해졌다. (c++17) <ul style="list-style-type: none"> <code>static_assert()</code> 메시지의 생략을 지원한다.

advanced template (c++11)	(c++11) <ul style="list-style-type: none"> 가변 템플릿 파라미터 팩이 추가되어 가변 인자 (...)와 같이 갯수와 타입이 정해지지 않은 템플릿 인자를 사용할 수 있다. <code>sizeof...</code>() 연산자가 추가되어 가변 템플릿에서 파라미터 팩의 인자수를 구할 수 있다. <code>extern</code>으로 템플릿을 선언할 수 있으며 템플릿 인스턴스 중복 생성을 없앨 수 있다. 템플릿 오른쪽 꺽쇠 괄호 파싱을 개선하여 템플릿 인스턴스화 시 >가 중첩되어 >>와 같이 되더라도 공백을 추가할 필요가 없어졌다. <p>(c++14)</p> <ul style="list-style-type: none"> <code>variable template</code>이 추가되어 변수도 템플릿으로 만들 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> 클래스 템플릿 인수 추론이 추가되어 함수 템플릿처럼 템플릿 인스턴스화시 타입을 생략할 수 있다. 클래스 템플릿 인수 추론 사용자 정의 가이드가 추가되어 클래스 템플릿 인수 추론 시 컴파일러에게 가이드를 줄 수 있다. 비타입 템플릿 인자에서 <code>auto</code>를 허용한다. Fold 표현식이 추가되어 가변 템플릿에서 파라미터 팩을 재귀적으로 반복하여 전개할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> 축약된 함수 템플릿이 추가되어 <code>auto</code> 타입을 사용할 수 있다. 사실 상 함수 템플릿의 간략한 표현이다. 비타입 템플릿 인자 규칙이 완화되어 실수 타입과 리터럴 타입을 사용할 수 있다. 클래스 템플릿 인수 추론 시 <code>initializer_list</code>인 경우가 개선되어 ,2,3<code>std::vector v1</code>처럼 템플릿 인자를 명시하지 않아도 된다.
concept (c++20)	(c++20) <ul style="list-style-type: none"> <code>concept</code>과 <code>requires</code>가 추가되어 템플릿 인자나 <code>auto</code>에 제약 조건을 줄 수 있다.

Coding convention enhancement

advanced type and literals ([c++11](#))

([c++11](#))

- 타입 카테고리를 수립하여 컴파일 타임 프로그래밍이나 템플릿 메타 프로그램이 시 코딩 컨벤션을 강화할 수 있다.
- `using`을 이용한 타입 별칭이 추가되어 `typedef`보다 좀 더 직관적인 표현이 가능해졌다.
- `nullptr` 리터럴이 추가되어 포인터에 안전한 코딩 컨벤션이 가능해졌다.
- `long long` 타입이 추가되어 최소 8byte 크기를 보장한다.
- `ll, ull, LL, ULL` 리터럴이 추가되어 `long long`용 정수형 상수를 제공한다.
- `char16_t, char32_t` 타입이 추가되어 UTF-16 인코딩, UTF-32 인코딩을 모두 지원한다.
- `u8"", u"", U"", u"(char), U"(char)` 리터럴이 추가되어 유니코드를 지원하는 `char16_t, char32_t` 타입용 문자 상수를 제공한다.
- `R"()"` 리터럴이 추가되어 개행이나 이스케이프 문자를 좀 더 편하게 입력할 수 있다.

([c++14](#))

- 이진 리터럴이 추가되어 `0b, 0B` 접두어로 이진수 상수를 표현할 수 있다.
- 숫자 구분자가 추가되어 `1'000'000`과 같이 작은 따옴표 '`'`를 숫자 사이에 선택적으로 넣을 수 있어 가독성이 좋아졌다.

([c++17](#))

- 16진수 부동 소수점 리터럴이 추가되어 `0xA, 9p11`과 같이 16진수로 실수를 표현할 수 있다.
- `u8"(char)` 리터럴이 추가되어 유니코드를 지원하는 1byte 크기의 문자 상수를 지원한다.

([c++20](#))

- `char8_t` 타입이 추가되어 UTF-8 인코딩 문자를 지원한다.
- 정수에서 2의 보수 범위를 보장한다.
- 사용자 정의 리터럴 인자 규칙에 `char8_t`가 추가되었다.

noexcept ([c++11](#))

([c++11](#))

- `noexcept`가 추가되어 함수의 예외 방출 여부를 보증하며 소멸자는 기본적으로 `noexcept`로 작동한다.
- `noexcept` 연산자가 추가되어 해당 함수가 `noexcept`인지 컴파일 타임에 검사할 수 있다.

([c++17](#))

- `noexcept`가 함수 유형에 포함되어 예외 처리에 대한 코딩 컨벤션을 좀 더 단단하게 할 수 있다.

explicit type conversion (c++11)	(c++11) <ul style="list-style-type: none"> • <code>explicit</code> 형변환 연산자가 추가되어 명시적으로 형변환할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> • <code>explicit(bool)</code>이 추가되어 특정 조건일 때만 <code>explicit</code>으로 동작하게 할 수 있다.
attribute (c++11)	(c++11) <ul style="list-style-type: none"> • <code>attribute</code>가 추가되어 컴파일러에게 부가 정보를 전달하는 방식을 표준화하였다. <p>(c++14)</p> <ul style="list-style-type: none"> • <code>[[deprecated]]</code>가 추가되어 소멸 예정인 것을 컴파일 경고로 알려준다. <p>(c++17)</p> <ul style="list-style-type: none"> • <code>[[fallthrough]]</code>가 추가되어 <code>switch()</code>에서 의도적으로 <code>break</code>를 생략하여 다음 case로 제어를 이동시킬 때 발생하는 컴파일 경고를 차단할 수 있다. • <code>[[nodiscard]]</code>가 추가되어 리턴값을 무시하지 않도록 컴파일 경고를 해준다. • <code>[[maybe_unused]]</code>가 추가되어 사용되지 않은 객체의 컴파일 경고를 차단할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> • <code>[[nodiscard]]</code>의 생성자 지원, <code>[[nodiscard("reason")]]</code>이 추가되었다. • <code>[[likely]], [[unlikely]]</code>가 추가되어 컴파일러에게 최적화 힌트를 줄 수 있다. • <code>[[no_unique_address]]</code>가 추가되어 아무 멤버 변수가 없는 객체의 크기를 최적화할 수 있다.

Coding convenience enhancement	
advanced namespace (c++11)	(c++11) <ul style="list-style-type: none"> • 인라인 네임스페이스가 추가되어 API 버전 구성이 편리해졌다. <p>(c++14)</p> <ul style="list-style-type: none"> • 단순한 중첩 네임스페이스가 추가되어 ::로 표현할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> • 인라인 네임스페이스와 단순한 중첩 네임스페이스를 결합하여 표현할 수 있다.

advanced initialization ([c++11](#))

([c++11](#))

- 중괄호 초기화를 제공하여 클래스, 배열, 구조체 구분없이 중괄호로 일관성있게 초기화를 할 수 있으며 초기화 패싱 오류를 해결하였다.
- 중괄호 복사 초기화로 함수 인수 전달, 리턴문 작성을 간결화할 수 있다.
- 중괄호 초기화시 인자의 암시적 형변환을 일부 차단하여 코딩 컨벤션이 개선되었다.
- `initializer_list`가 추가되어 `vector` 등 컨테이너의 초기 요소 추가가 간편해졌다.
- 멤버 선언부 초기화가 추가되어 non-static 멤버 변수의 초기화가 쉬어졌다.

([c++14](#))

- `non-static` 멤버 선언부 초기화 시 집합 초기화를 허용한다.

([c++20](#))

- 지명 초기화가 중괄호 집합 초기화 시 변수명을 지명하여 값을 초기화할 수 있다.
- 비트 필드 선언부 초기화가 추가되었다.
- `new[]`에서 중괄호 집합 초기화로 배열 크기 추론이 추가되어 배열 크기를 명시하지 않아도 된다.

advanced control statement ([c++11](#))

([c++11](#))

- `range-based for()`가 추가되어 컨테이너 요소의 탐색 처리가 쉬워졌다.

([c++17](#))

- 초기식을 포함하는 `if()`, `switch()`가 추가되어 함수 리턴 값을 평가하고 소멸하는 코드가 단순해졌다.

([c++20](#))

- `range-based for()`에서 초기식이 추가되었다.

advanced class (c++11)	(c++11) <ul style="list-style-type: none"> • <code>default</code>, <code>delete</code>가 추가되어 암시적으로 생성되는 멤버 함수의 사용 여부를 좀 더 명시적으로 정의할 수 있다. • <code>override</code>가 추가되어 가상 함수 오버라이딩 코딩 규약이 좀 더 단단해졌다. • <code>final</code>이 추가되어 가상 함수를 더 이상 오버라이딩 못하게 할 수 있고 강제적으로 상속을 제한할 수 있다. • 생성자 위임이 추가되어 생성자의 초기화 리스트 코드가 좀 더 간결해졌다. • 생성자 상속이 추가되어 부모 객체의 생성자도 상속받아 사용할 수 있어 자식 객체의 생성자 재정의 코드가 좀 더 간결해졌다. • 멤버 함수 참조 지정자가 추가되어 멤버 함수에 <code>&</code>, <code>&&</code>로 lvalue로 호출될 때와 rvalue로 호출 될 때를 구분하여 오버로딩을 할 수 있다.
auto decltype trailing return type (c++11)	(c++11) <ul style="list-style-type: none"> • <code>auto</code>와 <code>decltype()</code>이 추가되어 값으로부터 타입을 추론하여 코딩이 간편해졌다. 후행 리턴(trailing return)이 추가되어 함수 인자에 의존하여 리턴 타입을 결정하며 좀 더 동적인 함수 설계가 가능해졌다. <p>(c++14)</p> <ul style="list-style-type: none"> • <code>decltype(auto)</code>가 추가되어 <code>decltype()</code>의 () 내 표현식이 복잡할 경우 좀 더 간결하게 작성할 수 있다. • 리턴 타입 추론이 추가되어 후행 리턴 대신 <code>auto</code>나 <code>decltype(auto)</code>를 사용할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> • <code>auto</code>의 중괄호 초기화 특수 추론 규칙이 개선되어 <code>auto a1; </code> 시 <code>initializer_list</code>가 아니라 <code>int</code>로 추론된다.
scoped enum (c++11)	(c++11) <ul style="list-style-type: none"> • 범위 있는 열거형(scoped enum)이 추가되어 이름 충돌 회피가 쉬워졌고 암시적 형변환을 차단하며 전방 선언도 지원한다. <p>(c++17)</p> <ul style="list-style-type: none"> • 열거형의 중괄호 직접 초기화를 허용하여 암시적 형변환을 차단하는 사용자 정의 열거형의 사용이 좀 더 쉬워졌다. <p>(c++20)</p> <ul style="list-style-type: none"> • <code>using enum</code>이 추가되어 열거형의 이름 없이 열거자를 유효 범위 내에서 사용할 수 있다.

lambda expression, closure (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> 람다 표현식이 추가되어 1회용 익명 함수를 만들 수 있다. <p>(c++14)</p> <ul style="list-style-type: none"> 람다 캡쳐 초기화가 추가되어 람다 표현식 내에서 사용하는 임의 변수를 정의하여 사용할 수 있다. 일반화된 람다 표현식이 추가되어 <code>auto</code>를 받아 마치 함수 템플릿처럼 사용할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> 람다 캡쳐 시 <code>*this</code>가 추가되어 객체 자체를 복제하여 사용할 수 있다. <code>constexpr</code> 람다 표현식이 추가되어 람다 표현식도 컴파일 타임 함수로 만들 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> 람다 표현식에서 템플릿 인자를 지원한다. 람다 캡쳐에서 파라미터 팩을 지원한다. 람다 캡쳐에서 구조화된 바인딩을 지원한다. 상태없는 람다 표현식의 기본 생성과 복사 대입을 지원한다. 미평가 표현식에서도 람다 표현식을 허용하기 때문에 <code>decltype()</code> 안에서 사용할 수 있다.
advanced variables (c++17)	<p>(c++17)</p> <ul style="list-style-type: none"> 인라인 변수가 추가되어 헤더 파일에 정의된 변수를 여러 개의 cpp에서 <code>#include</code> 하더라도 중복 정의없이 사용할 수 있다. 또한 클래스 정적 멤버 변수를 선언부에서 초기화할 수 있다.
structured bindings (c++17)	<p>(c++17)</p> <ul style="list-style-type: none"> 구조화된 바인딩(structured bindings)이 추가되어 배열, <code>pair</code>, <code>tuple</code>, <code>class</code> 등 내부 요소나 멤버 변수에 쉽게 접근할 수 있다.
advanced operator (c++20)	<p>(c++20)</p> <ul style="list-style-type: none"> 삼중 비교 연산자가 추가되어 비교 연산자 구현이 간소화되었다. 삼중 비교 연산자를 <code>default</code>로 정의할 수 있다. 비트 쉬프트 연산자의 기본 비트가 표준화되어 « 1은 곱하기 2의 효과가 있는 비트(즉, 0)으로 채워지고 » 1은 나누기 2의 효과가 있는 비트 (즉, 양수면 0, 음수면 1)로 채워진다.

module (c++20)	(c++20) <ul style="list-style-type: none"> 모듈이 추가되어 전처리 사용 방식을 개선하여 컴파일 속도를 향상시키고, #include 순서에 따른 종속성 문제, 선언과 정의 분리 구성의 불편함, 기호 충돌 문제를 해결하였다.
coroutine (c++20)	(c++20) <ul style="list-style-type: none"> 코루틴이 추가되어 함수의 일시 정지 후 재개가 가능해졌다.
misc	(c++11) <ul style="list-style-type: none"> alignas(), alignof()가 추가되어 메모리 정렬 방식을 표준화하였다. 가변 매크로가 추가되어 c언어와 호환성이 높아졌다. 멤버의 sizeof() 시 동작이 개선되어 객체를 인스턴스화 하지 않아도 객체 멤버의 크기를 구할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> __has_include가 추가되어 #include 하기 전에 파일이 존재하는지 확인할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> __VA_OPT__가 추가되어 가변 인자가 있을 경우에는 괄호 안의 값으로 치환하고 없을 경우에는 그냥 비워둔다. __has_cpp_attribute() 매크로 함수가 추가되어 c++11부터 추가된 attribute가 지원되는지 확인할 수 있다. 언어 지원 테스트 매크로가 추가되어 c++11부터 추가된 언어 기능을 지원하는지 테스트할 수 있다.

deprecated & removed

(c++11)

- 동적 예외 사양은 `deprecated`되었다. 예외를 나열하는 것 보다 `noexcept`로 예외를 방출하느냐 안하느냐만 관심을 둔다.
- `export` 템플릿은 제대로 구현한 컴파일러는 드물고 세부 사항에 대한 의견이 일치하지 않아 c++11부터 완전히 remove 되었다.

(c++17)

- 동적 예외 사양 관련해서 `throw()`가 `deprecated`되었다. 이제 `noexcept`만 사용해야 한다.
- `&&`, `&=` 등이 특수기호가 없는 인코딩을 사용하는 곳을 위해 제공했던 trigraph가 remove되었다.
- 변수를 CPU 레지스터에 배치하도록 힌트를 주는 `register` 가 `deprecated`되었다.
- `bool`의 증감 연산이 `deprecated`되었다.

(c++20)

- 람다 캡쳐에서 [=] 사용 시 `this`의 암시적 캡쳐가 `deprecated`되었으므로 명시적으로 작성해야 한다.
- `volatile` 일부가 `deprecated`되었다.

2 Intermediate

2.1 Temporary

```
1 class Point{
2     int x,y;
3     public:
4
5     Point(int x, int y) : x(x), y(y) { std::cout << "Point(int,int)" << std::endl; }
6     ~Point() { std::cout << "~Point()" << std::endl; }
7 };
8
9 Point pt(1,2); // --> 일반 객체
10 Point (1,2); // --> 임시 객체
11
12 pt.x = 10;           // ok
13 Point(1,2).x = 10; // error (rvalue라고 부름)
14 Point(1,2).set(10,20) // ok (멤버함수는 불러와지므로 상수는 아님)
15
16 Point *p1 = &pt;      // ok
17 Point *p2 = &Point(1,2) // error (임시객체는 포인터로 가르킬 수 없다)
18
19 Point& r1 = pt;      // ok
20 Point& r2 = Point(1,2); // error(임시객체는 참조로 가르킬 수 없다)
21 Point&& r3 = Point(1,2); // ok (rvalue reference 문법)
22
23 const Point &r4 = Point(1,2) // ok(상수 참조는 가능, 일반 객체로 승격되는 효과)
```

객체를 함수 인자로만 사용한다면 임시객체로 전달하는게 효율적일 수 있다. (`const` reference로 받아야만 함)

```
1 void foo(const Point& pt) { std::cout << "foo" << std::endl; }
2
3 int main() {
4     Point pt(1,2);
5     foo(pt);          // pt는 foo에 넣기 위해서만 생성한 객체이므로 함수가 불린 후 바로 파괴되는게 좋다.
6
7     foo(Point(1,2)) // 임시 객체를 활용하자
8     foo( {1, 2} )   // 이렇게 전달하는 것도 가능하다. 컴파일러가 Point1,2로 바꿔줌
9     std::cout << "-----" << std::endl;
10 }
```

```
1 void foo(const std::string& s) { }
2 void goo(std::string_view s) { } // call-by-value임에 유의!
3
4 int main() {
5     foo("Practice make perfect"); // 컴파일러에서 string("Practice make perfect")로 변환해줌
6
7     goo("Practice make perfect"); // string_view는 문자열의 복사본을 생성하지 않고 이미 상수 메모리의
8     존재하는 문자열을 가르킨다.
9 }
```

2.1.1 return-by-value vs return-by-reference

```
1 Point pt(1,2);
2 Point f3() { return pt; }
3 Point& f4() { return pt; }
4
5 Point& f5() {
6     Point pt(1,2);
7     return pt;    // error. 지역객체를 return-by-reference하면 안됨!
8 }
9
10 int main() {
```

```
11     f3().x = 10; // error. (return-by-value는 복사본이 생성되어 임시객체이므로 rvalue임)
12     f4().x = 10; // ok. pt.x = 10
13 }
```

```
1 class Counter {
2     int count{0};
3
4     Counter& increment() { //return-by-reference로 하는걸 잊으면 안됨!
5         ++count;
6         return *this;
7     }
8     int get() const { return count; }
9 }
10
11 int main() {
12     Counter c;
13     c.increment().increment().increment();
14     std::cout << c.get() << std::endl;
15 }
```

2.1.2 temporary and casting

```
1 struct Base {
2     int value = 10;
3
4     Base() = default;
5     Base(const Base&b) : value(b.value)
6     { std::cout << "copy constructor" << std::endl; }
7 }
8
9 struct Derived : public Base {
10     int value = 10;
11 }
12
13 int main() {
14     Derived d;
15
16     std::cout << d.value << std::endl; // 20
17     std::cout << static_cast<Base&>(d).value << std::endl; // 10 good. 참조 캐스팅을 통해 d를 Base로
18     생각하게 하여 값을 가져온다.
19     std::cout << static_cast<Base>(d).value << std::endl; // 10 no. Base 클래스의 복사본이 생성되며
20     거기서 값을 가져온다.
21
22     static_cast<Base&>(d).value = 100;
23     static_cast<Base>(d).value = 100; // error. 캐스팅은 항상 reference로 하자!
}
```

2.2 Conversion

이번 섹션에서는 객체 변환에 대한 다양한 문법과 기법에 대해 설명한다.

- 변환 연산자, 변환 생성자, `explicit` 생성자의 개념
- safe bool 개념과 `explicit` 변환 연산자에 대해 살펴본다.
- `nullptr`과 return type resolver 기술
- temporary proxy 기술
- lambda expression과 함수 포인터 변환

2.2.1 Conversion constructor, conversion operator

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5 };
6
7 int main() {
8     int pn;          // primitive type
9     Int32 un;      // user type 값을 지정해주지 않아도 쓰레기값이 아닌 0으로 초기화된다
10
11    pn = un;
12 }
```

int를 대체하기 위한 Int32 클래스에 대해 알아보자. 새로 정의한 타입은 기존에 타입과도 호환이 되는게 좋기 때문에 pn = un과 같이 서로 변환이 가능해야 한다.

pn = un이 호출된 순간 컴파일러는 un.operator int() 함수를 찾게 되는데 이러한 연산자를 **변환 연산자 (conversion operator)**라고 한다.

```
1 operator TYPE() { 변환 연산자
2     return value;
3 }
```

변환 연산자는 반환 타입이 함수 이름에 포함되어 있으므로 반환 타입을 표기하지 않는다.

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5     operator int() const { return value; } const를 추가하여 상수 객체 또한 동작하도록 해준다
6 };
7
8 int main() {
9     int pn;
10    Int32 un;
11    const Int32 un2;
12
13    pn = un;
14    pn = un2;
15 }
```

다음으로 un = pn과 같이 반대의 경우를 보자. 이런 경우 컴파일러는 un.operator=(int) 대입연산자가 존재하는지 먼저 검사한다. 만약 없다면 Int32(int)와 같이 default 대입 연산자가 있는지 검사한다.

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5     Int32(int n) : value(n) {} 변환 생성자
6     operator int() const { return value; }
7 };
8
9 int main() {
10    int pn;          // primitive type
11    Int32 un;      // user type 값을 지정해주지 않아도 쓰레기값이 아닌 0으로 초기화된다
12    const Int32 un2;
13
14    pn = un;
15    pn = un2;
16    un = pn; // 변환 생성자 호출!
17 }
```

변환 생성자에 대해 좀 더 자세히 알아보자

```

1 class Int32 {
2     int value;
3     public:
4     Int32(int n) : value(n) {}
5 };
6
7 int main() {
8     Int32 n1(3);      // 1. direct initialization
9     Int32 n2 = 3;    // 2. copy initialization
10    Int32 n3{3};     // 3. direct init. (c++11)
11    Int32 n4 = {3}; // 4. copy init. (c++11)
12
13    n1 = 3; // conversion (int -> Int32)
14 }
```

변환 생성자가 있는 경우 위와 같이 변환 이외에도 4가지 초기화 코드를 만들 수 있다. 이 중 2번에 대해 자세히 알아보자.

`Int32 n2 = 3`이 호출되면 컴파일러는 이를 `Int32 n2 = Int32(3)`으로 변환한다. 이는 임시 객체(temporary)이므로 **c++98** 시절까지는 복사 생성자를 통해 `n2`에 대입되지만 **c++11** 이후부터는 `move` 생성자를 통해 `n2`로 대입된다. 하지만 대부분의 컴파일러에서 최적화 옵션을 켜면 임시 객체 생성이 제거된다.

```

1 class Int32 {
2     int value;
3     public:
4     Int32(int n) : value(n) {}
5     Int32(const Int32&) = delete; // 복사 생성자 제거
6 };
7
8 int main() {
9     Int32 n1(3);
10    Int32 n2 = 3; // error! c++14까지는 에러가 발생하지만 c++17부터는 문법적으로 복사 생성을 하지 않기
11    때문에 ok
12    Int32 n3{3};
13    Int32 n4 = {3};
14
15    n1 = 3;
16 }
```

다음으로 마지막 줄의 `n1 = 3`을 살펴보자. 이는 컴파일러에 의해 `n1 = Int32(3)`으로 변환된다. 즉, 임시 객체가 생성되고 디폴트 대입 연산자를 사용해서 `n1`에 대입되는 형태이다. 대부분의 컴파일러가 최적화를 통해 임시 객체 생성이 제거되지만 **디폴트 대입 연산자는 반드시 존재해야 한다.**

```

1 class Int32 {
2     int value;
3     public:
4     Int32(int n) : value(n) {}
5     Int32(const Int32&) = delete; // 복사 생성자 제거
6     Int32& operator=(const Int32&) = delete; // 디폴트 대입 연산자 제거
7 };
8
9 int main() {
10    Int32 n1(3);
11    //Int32 n2 = 3;
12    Int32 n3{3};
13    Int32 n4 = {3};
14
15    n1 = 3; // error. 디폴트 대입 연산자가 없으면 모든 버전에 대해 에러가 발생한다.
16 }
```

2.2.2 explicit constructor

```

1 class Vector{
2     public:
3     Vector(int size) {}
```

```

4   };
5   void foo(Vector v) {}
6   int main() {
7     Vector v1(3);
8     Vector v2 = 3;
9     Vector v1{3};
10    Vector v1 = {3};

11
12    v1 = 3; // 논리적으로 벡터에 3을 넣는게 맞지 않음!
13    foo(3); // ok but no. Vector v = 3의 형태로 복사생성자가 호출됨
14 }

```

explicit 생성자를 사용하면 생성자가 암시적 변환의 용도로 사용될 수 없게 한다. 이런 경우 직접 초기화(direct initialization)만 가능하고 복사 초기화(copy initialization)은 사용할 수 없다.

```

1 class Vector{
2   public:
3     explicit Vector(int size) {}
4   };
5   void foo(Vector v) {}
6   int main() {
7     Vector v1(3);
8     Vector v2 = 3; // error
9     Vector v1{3};
10    Vector v1 = {3}; // error

11
12    v1 = 3; // error
13    foo(3); // error
14 }

```

`explicit`을 사용할 때는 클래스에 따라 `explicit`을 사용할 지 잘 판단해야 한다.

```

1 void f1(Int32 n) {}
2 void f2(Vector v) {}

3
4 f1(3); // ok. 논리적으로 맞으므로 Int32 생성자에는 explicit를 붙일 필요 없음
5 f2(3); // ok but no. 논리적으로 맞지 않으므로 Vector 생성자에는 explicit를 붙여야함

```

2.2.3 explicit conversion operator

객체의 유효성을 `if`문을 통해 비교하고 싶다고 하자.

```

1 class Machine {
2   int data = 10;
3   bool state = true;
4 };
5 int main() {
6   Machine m;
7   if(m) {} // 객체의 유효성을 if문을 통해 비교하고자 한다
8 }

```

컴파일을 해보면 'Machine을 `bool`로 변환할 수 없다'는 에러 구문이 발생하는데 이는 곧 `Machine을 bool로만 변화할 수 있으면 if문이 동작한다는 얘기와 같다.`

```

1 class Machine {
2   int data = 10;
3   bool state = true;
4   public:
5     operator bool() { return state; } // bool 연산자
6   };
7
8   int main() {
9     Machine m;
10    if(m) {} // ok
11 }

```

`if(m)` 구문은 이제 정상적으로 작동하지만 `operator bool()`은 side effect가 많아서 사용에 주의해야 한다. 예를 들어 다음과 같은 side effect가 발생한다.

```
1 class Machine {
2     int data = 10;
3     bool state = true;
4     public:
5         operator bool() { return state; }
6     };
7
8     int main() {
9         Machine m;
10
11     bool b1 = m; // ok
12     bool b2 = static_cast<bool>(m); // ok
13     m << 10; // 문법이 이상하지만 컴파일 된다. 즉, 버그의 원인이 될 수 있다.
14     if(m) { ... }
15 }
```

이러한 문제를 해결하기 위해 c++11부터 생성자 뿐만 아니라 변환 연산자도 `explicit`을 붙일 수 있다. 이러한 기술을 **safe bool**이라고 부른다.

```
1 class Machine {
2     int data = 10;
3     bool state = true;
4     public:
5         explicit operator bool() { return state; }
6     };
7
8     int main() {
9         Machine m;
10
11     bool b1 = m; // error
12     bool b2 = static_cast<bool>(m); // ok. 명시적 변환이므로
13     m << 10; // error
14     if(m) { ... }
15 }
```

c++98부터 `explicit` 변환 생성자 구문을 제공하였으나 c++11이 되면서 `explicit` 변환 연산자 문법이 추가되었다. 그리고 c++20에는 `explicit(bool)` 문법이 제공되었다. `explicit(bool)` 문법에 대해 알아보자.

2.2.4 `explicit(bool)`, c++20

```
1 template<class T>
2 class Number {
3     T value;
4     public:
5         explicit(true) Number(T v) : value(v) {}
6     };
7
8     int main() {
9         Number n1 = 10; // error
10        Number n2 = 3.4; // error. explicit이 true이므로 에러 발생. 만약 explicit(false)이면 컴파일이
11        정상적으로 된다
11 }
```

위 코드에서 `explicit(true)`를 하면 `explicait` 키워드를 사용한다는 의미이고 반대로 `explicit(false)`를 하면 사용하지 않는다는 의미이다. 만약 정수형일 때만 `explicit`을 사용하지 않고 `float`일 때는 `explicit`을 사용하고 싶으면 어떻게 해야 할까?

```
1 template<class T>
2 class Number {
3     T value;
4     public:
5         explicit(!std::is_integral_v<T>)
```

```

6     Number(T v): value(v) {}
7 };
8 int main() {
9     Number n1 = 10; // ok. explicit(false)이므로
10    Number n2 = 3.4; // error. explicit(true)이므로
11 }

```

2.2.5 Conversion example: nullptr

`nullptr`은 포인터 초기화 시 0 대신 사용하는 C++ 키워드이다. 원래 boost 라이브러리에 있는 도구를 C++11을 만들면서 표준에 추가되었다. 이번 섹션에서는 boost에 있는 `nullptr`을 구현하면서 `nullptr`의 개념에 대해 다시 한번 숙지해보자.

```

1 void foo(int* p) {}
2 void goo(char* p) {}

3
4 struct nullptr_t {
5     template<class T>
6     constexpr operator T*() const { return 0; }
7 };
8 nullptr_t xnullptr;

9
10 int main() {
11     foo(xnullptr);
12     goo(xnullptr); // nullptr은 이미 키워드이므로 임의의 xnullptr을 정의하였다.
13 }

```

위 코드 예제와 유사하게 실제 `nullptr`의 타입도 `std::nullptr_t`이다.

2.2.6 Conversion example: return type resolver

Return type resolver는 좌변을 보고 우변의 반환 타입을 자동으로 결정하는 기술을 말한다. 다음과 같은 예제 코드를 보자.

```

1 template<class T>
2 T* Alloc(std::size_t sz) {
3     return new T[sz];
4 }
5
6 int main() {
7     int* p1 = Alloc<int>(10);
8     double* p2 = Alloc<double>(10); // <double>과 같은 꺽쇠를 사용하지 않고 바로 Alloc을 사용할 수는
9     없을까?

```

`Alloc(10)`으로 바로 받기 위해서는 `Alloc`이 함수가 아니라 구조체여야 한다.

```

1 struct Alloc {
2     std::size_t size;
3     Alloc(std::size_t sz) : size(sz) {}
4
5     template<class T>
6     operator T*() { return new T[size]; }
7 };
8
9 int main() {
10    int* p1 = Alloc(10);
11    double* p2 = Alloc(10); // 변환연산자에 의해 자동으로 타입이 결정되어 할당된다.

```

2.2.7 Lambda expression and conversion

일반적으로 람다 표현식은 `auto` 변수에 담아서 사용하지만 함수 포인터로도 람다 표현식을 담을 수 있다.

```

1 int main() {

```

```

2     auto f1 = [](int a, int b) { return a + b; }
3     int (*f2)(int, int) = [](int a, int b) { return a + b; } // 람다 표현식은 임시객체를 만들어주므로
4         임시객체.operator 함수포인터()가 호출되어 함수포인터에 할당된다.
}

```

2.3 Constructor

이번 섹션에서는 생성자의 호출 원리에 대해 살펴본다. 다음과 같은 4개의 구조체를 보자.

```

1 struct BM {
2     BM() { cout << "BM()" << endl; }
3     ~BM() { cout << "~BM()" << endl; }
4 };
5 struct DM {
6     DM() { cout << "DM()" << endl; }
7     DM(int) { cout << "DM(int)" << endl; }
8     ~DM() { cout << "~DM()" << endl; }
9 };
10 struct Base {
11     BM bm;
12     Base() { cout << "Base()" << endl; }
13     Base(int a) { cout << "Base(int)" << endl; }
14     ~Base() { cout << "~Base()" << endl; }
15 };
16 struct Derived : public Base {
17     DM dm;
18     Derived() { cout << "Derived()" << endl; }
19     Derived(int a) { cout << "Derived(int)" << endl; }
20     ~Derived() { cout << "~Derived()" << endl; }
21 };
22
23 int main() {
24     Derived d1; // call Derived::Derived()
25     Derived d2(7); // call Derived::Derived(int)
26 }

```

d1, d2을 호출하는 순간 컴파일러에 의해 자동 생성된 코드가 실행된다.

```

1 struct Base {
2     BM bm;
3     Base() : bm() { cout << "Base()" << endl; }
4     Base(int a) : bm() { cout << "Base(int)" << endl; }
5     ~Base() { cout << "~Base()" << endl; bm. BM(); }
6 };
7 struct Derived : public Base {
8     DM dm;
9     Derived() : Base(), dm() { cout << "Derived()" << endl; }
10    Derived(int a) : Base(), dm() { cout << "Derived(int)" << endl; }
11    ~Derived() { cout << "~Derived()" << endl; dm. DM(); Base(); }
12 };
13
14 int main() {
15     Derived d1; // call Derived::Derived()
16 }

```

생성자, 소멸자 호출자를 정확히 명시해주지 않아도 컴파일러가 **기반 클래스 및 멤버 데이터의 생성자(소멸자)**를 호출해주는 코드를 생성해준다.

다음으로 호출 순서를 정확히 아는 것이 중요하다. d1을 호출하면 **bm() → Base() → dm() → Derived()**가 순서대로 호출된다. 임의로 코드의 위치를 바꿔도 **사용자가 순서대로 호출 순서를 변경할 수 없다.**

또한, 컴파일러가 생성한 코드는 항상 디폴트 생성자를 호출한다. **기반 클래스나 멤버 데이터에 디폴트 생성자가 없는 경우 반드시 사용자가 디폴트가 아닌 다른 생성자를 호출하는 코드를 작성해야 한다.**

```

1 struct BM {
2     BM() { cout << "BM()" << endl; }
3     ~BM() { cout << "~BM()" << endl; }

```

```

4   };
5   struct DM {
6     //DM() { cout << "DM()" << endl; }
7     DM(int) { cout << "DM(int)" << endl; }
8     ~DM() { cout << "~DM()" << endl; }
9   };
10  struct Base {
11    BM bm;
12    //Base() { cout << "Base()" << endl; }
13    Base(int a) { cout << "Base(int)" << endl; }
14    ~Base() { cout << "~Base()" << endl; }
15  };
16  struct Derived : public Base {
17    DM dm;
18    Derived() : Base(0), dm(0) { cout << "Derived()" << endl; } // Base(0), dm(0)을 호출하지 않으면 에러
19    발생!
20    Derived(int a) : Base(0), dm(0) { cout << "Derived(int)" << endl; }
21    ~Derived() { cout << "~Derived()" << endl; }
22  };
23
24  int main() {
25    Derived d1;
26    Derived d2(7);
27 }

```

2.3.1 Base from member idioms

다음과 같이 버퍼를 사용하는 스트림 예제 코드를 보자.

```

1  class Buffer {
2    public:
3      Buffer(std::size_t sz) { cout << "initialize buffer" << endl; }
4      void use() { cout << "use buffer" << endl; }
5  };
6
7  class Stream {
8    public:
9      Stream(Buffer& buf) { buf.size(); }
10 };
11
12 int main() {
13   Buffer buf(1024);
14   Stream s(buf);
15 }

```

위 코드는 정상적으로 initialize buffer를 통해 버퍼를 초기화한 후 use buffer가 호출되어 버퍼를 사용한다.
다음으로 buffer를 멤버함수로 두고 싶은 경우를 생각해보자.

```

1  class Buffer {
2    public:
3      Buffer(std::size_t sz) { cout << "initialize buffer" << endl; }
4      void use() { cout << "use buffer" << endl; }
5  };
6  class Stream {
7    public:
8      Stream(Buffer& buf) { buf.size(); }
9  };
10
11 class StreamWithBuffer : public Stream {
12   Buffer buf(1024);
13   public:
14     StreamWithBuffer() : Stream(buf) {}
15 };
16
17 int main() {

```

```
18     StreamWithBuffer swf; // error. use buffer, initialize buffer 순으로 잘못 호출됨!
19 }
```

위 코드는 초기화되지 않은 버퍼를 사용하는 문제가 발생한다. 즉, **멤버 데이터보다 기반 클래스의 생성자가 먼저 호출되는 문제가 발생한다.**

```
1 class StreamWithBuffer : public Stream {
2     Buffer buf(1024);
3     public:
4     StreamWithBuffer() : Stream(buf), buf(1024) {} // 컴파일러에 의해 buf(1024)가 늦게 호출된다
5 };
```

이를 해결하는 방법은 다중 상속으로 해결해야 한다.

```
1 class StreamBuffer {
2     protected:
3     Buffer buf(1024);
4 };
5
6 class StreamWithBuffer : public StreamBuffer, public Stream { // 다중 상속을 통해 문제를 해결
7     public:
8     StreamWithBuffer() : StreamBuffer(), Stream(buf) {}
9 };
```

위 기술은 c++ 초기 설계 당시 ostream을 만들 때 사용한 방법이며 "**Base from member(c++ idioms)**"라는 이름을 가지고 있다.

2.3.2 Constructor and virtual function

가상함수를 호출하는 시점에 따라 Dervied 클래스의 함수를 호출하거나 Base의 함수를 호출하는 동작이 달라진다.

```
1 class Base{
2     public:
3     Base() { vfunc(); }    // Base vfunc
4     void foo() { vfunc(); } // Derived vfunc
5     virtual void vfunc() { cout << "Base vfunc()" << endl; }
6 };
7
8 class Derived : public Base {
9     int data{10};
10    public:
11    virtual void vfunc() override{ cout << "Derived vfunc()" << data << endl; }
12 };
13
14 int main() {
15     Derived d;
16     d.foo(); // Derived의 vfunc가 실행된다.
17 }
```

생성자에서는 가상 함수가 동작하지 않는다. 왜 이렇게 설계되었을까? Dervied 생성자는 컴파일러를 통해 다음과 같이 생성된다.

```
1 class Derived : public Base {
2     int data{10};
3     public:
4     Derived() : Base(), data(10)
5     virtual void vfunc() override{ cout << "Derived vfunc()" << data << endl; }
6 };
```

Base()가 먼저 호출되기 때문에 data의 정보를 사용하는 Derived의 vfunc를 호출하면 잘못된 정보를 호출하게 된다. 따라서 Base 생성자에서는 Base vfunc가 호출된다.

2.4 This call

```

1   class Point{
2     int x{0};
3     int y{0};
4     public:
5     void set(int a, int b) {
6       x=a; y=b;
7     }
8   };
9
10  int main(){
11    Point pt1;
12    Point pt2;
13
14    pt1.set(10,20);
15    pt2.set(10,20);
16 }
```

pt1, pt2를 생성하면 메모리 공간에 **멤버 변수가 객체 당 하나씩 생성된다.** 그렇다면 멤버 함수 set 또한 객체 당 하나씩 생성될까? 그렇지 않다. 멤버 함수는 객체가 여러개 생성되어도 **메모리에 한 개만** 만들어져 있다.

그렇다면 set 함수 인자가 a, b 밖에 없는데 x가 어떤 객체의 멤버인지 어떻게 알 수 있을까? 컴파일러가 이런 문제를 해결하기 위해 다음과 같이 컴파일 해준다.

```

1   class Point{
2     int x{0};
3     int y{0};
4     public:
5     void set(Point* this, int a, int b) {
6       this->x=a; this->y=b;
7     }
8   };
9   int main(){
10    Point pt1;
11    Point pt2;
12
13    set(pt1, 10,20);
14    set(pt2, 10,20);
15 }
```

사용자가 2개의 인자로 set를 구성했다하더라도 컴파일러가 객체 포인터 주소 인자가 같이 전달되도록 3개의 인자로 구성된 set 함수가 완성된다. 이러한 기술을 **this call**이라고 한다.

주의할 점은 실제 함수 인자가 전달되는 방식과 객체 주소가 전달되는 방식은 어셈블리 레벨에서는 약간 차이가 있다. 컴파일러마다 구현하는 방식도 다르니 위 예제 코드는 저런 개념으로 전달된다 정도만 숙지하면 된다.

static 멤버 함수는 this call이 적용되지 않는다.

```

1   class Point{
2     int x{0};
3     int y{0};
4     public:
5     void set(int a, int b) {
6       x=a; y=b;
7     }
8
9     static void foo(int a) {
10      x=a;
11    }
12  };
13
14  int main(){
15    Point pt1;
16    Point pt2;
17
18    pt1.set(10,20);
19    pt2.set(10,20);
20
21    Point::foo(10); // static 멤버함수는 객체 주소를 전달하지 않는다.
```

```
22     pt1.foo(10); // 실제로는 객체 주소 없이 Point::foo(10)으로 변환되어 실행된다.
23 }
```

2.5 Member function pointer

멤버 함수를 함수 포인터로 가르킬 수 있을까? 다음 예제를 보자.

```
1 class X {
2     public:
3         void mf1(int a) {}
4         static void mf2(int a) {}
5     };
6
7     void foo(int a) {}
8
9     int main() {
10         void(*f1)(int) = &foo;           // ok
11         void(*f2)(int) = &X::mf1; // error. this call에 의해 파라미터 개수 맞지 않음!
12         void(*f3)(int) = &X::mf2; // ok. static 멤버 함수는 this call이 없으므로 적용 가능
13 }
```

일반 함수 포인터에 **멤버 함수의 주소를 담을 수 없다**. 하지만 **static** 멤버 함수의 주소는 담을 수 있다. 그렇다면 멤버 함수의 주소를 담으려면 어떻게 해야 할까?

```
1 class X {
2     public:
3         void mf1(int a) {}
4         static void mf2(int a) {}
5     };
6
7     void foo(int a) {}
8
9     int main() {
10         void(*f1)(int) = &foo;
11         //void(*f2)(int) = &X::mf1;
12         void(*f3)(int) = &X::mf2;
13
14         void(X::*f2)(int) = &X::mf1; // ok. 멤버 함수 주소는 이렇게 담아야 한다.
15 }
```

일반 함수 포인터는 `void(*f1)(int)= foo` 또는 `&foo`를 입력해도 함수 주소로 암시적 변환이 가능하지만 멤버 함수 포인터는 `void(X::*f2)= X::mf1`으로 하면 컴파일 에러가 발생하므로 반드시 `&X::mf1`으로 주소를 넣어줘야 함에 유의한다.

멤버 함수 포인터는 일반 함수 포인터처럼 호출할 수 있을까?

```
1 f1(10); // ok
2 f2(10); // error. 객체가 필요하다.
```

멤버 함수 포인터는 객체가 있어야 사용 가능하다.

```
1 X obj;
2 obj.f2(10); // error. f2라는 멤버를 찾게 된다.
3
4 // pointer to member 연산자 사용
5 obj.*f2(10); // error. 연산자 우선순위 문제로 괄호가 먼저 계산된다.
6 (obj.*f2)(10); // ok
```

`.*`는 하나의 연산자 역할을 하며 **pointer to member operator**라고 부른다. 객체가 포인터인 경우 `->*`을 사용하면 된다.

```
1 (obj.*f2)(10); // ok
2 (pobj->*f2)(10); // ok
```

2.5.1 std::invoke, std::mem_fn

멤버 함수 포인터를 사용하는 형태가 너무 복잡해보인다. 이를 일반 함수 포인터처럼 조금 더 쉽게 호출할 수는 없을까?

```
1 class X {
2     public:
3         void mf1(int a) {}
4         static void mf2(int a) {}
5     };
6
7     void foo(int a) {}
8
9     int main() {
10        void(*f1)(int) = &foo;
11        void(X::*f2)(int) = &X::mf1;
12
13        X obj;
14
15        f1(10);          // 일반 함수 포인터 사용
16        (obj.*f2)(10); // 멤버 함수 포인터 사용
17        f2(&obj, 10);   // 위 모양이 너무 복잡해서 이렇게 사용할 수는 없을까? 실제 논의가 있었다.
18        // 이를 uniform call syntax라고 한다.
19    }
```

하지만 **uniform call syntax**는 컴파일러가 어셈블리 레벨을 많이 바꿔야 하기 때문에 채택되지 않았다. 대신 STL에서는 몇 가지 도구를 제공한다.

`std::invoke`는 c++17부터 나왔으며 일반 함수 포인터와 멤버 함수 포인터(정확히는 **callable object**)를 정확히 동일한 방법으로 호출할 수 있다.

```
1 #include <functional>
2
3     std::invoke(f1, 10);
4     std::invoke(f2, obj, 10);
5     std::invoke(f2, &obj, 10);
```

또는 `std::mem_fn`을 사용해도 된다. 이는 c++11부터 나왔으며 멤버 함수 주소를 인자로 받아서 함수 주소를 담은 래퍼 객체를 반환한다.

```
1 #include <functional>
2
3     auto f3 = std::mem_fn(&X::mf1); // 반드시 auto로 받아야 한다.
4     f3(obj, 10);
5     f3(&obj, 10);
```

2.6 Member data pointer

일반 변수의 포인터처럼 멤버 변수의 포인터도 만들 수 있을까?

```
1 struct Point {
2     int x;
3     int y;
4 };
5
6     int main() {
7         int num=0;
8         int *p1 = &num; // ok
9
10        int Point::*p2 = &Point::y; // ok. 멤버 변수의 포인터는 이렇게 가르켜야 한다.
11    }
```

p1은 num 변수의 메모리 주소를 가지고 있다. 그런데 Point 타입의 객체가 존재하지 않는데 p2는 무엇을 담고 있을까? 대부분의 컴파일러는 Point 구조체 안에서 y의 offset을 담고 있다. 이는 공식 표준은 아니지만 대부분의 컴파일러가 이렇게 구현되어 있다.

멤버 변수 포인터는 다음과 같이 사용한다.

```
1 *p1 = 10; // ok
2 *p2 = 10; // error
3
4 Point pt;
5 pt.*p2 = 10; // ok. pt.y = 10
6 // *(&pt + p2) = 10; p2만큼 오프셋 주소를 더하여 그 곳의 변수에 10을 넣는다.
```

멤버 변수 포인터도 `invoke`를 통해 호출할 수 있다.

```
1 std::invoke(p, obj) = 10; // ok. obj.y = 10. 일반 함수 포인터처럼 사용할 수 있다.
2 int n = std::invoke(p, obj); // ok. 데이터를 꺼내는 것도 가능하다
```

`std::invoke`는 일반 포인터, 함수, 멤버 함수 포인터, 멤버 변수 포인터, 람다 표현식을 전부 일관되게 동작할 수 있게 해준다. 이런 타입들을 **callable type**라고 한다.

2.7 Implement custom max

이번 섹션에서는 임의의 `max` 함수를 작성해보자.

```
1 #include<iostream>
2 #include<string>
3
4 template<class T>
5 const T& mymax(const T& obj1, const T& obj2) {
6     return obj1 < obj2 ? obj2 : obj1;
7 }
8
9 int main() {
10    std::string s1 = "abcd";
11    std::string s2 = "xyz";
12
13    auto ret1 = mymax(s1, s2);
14    std::cout << ret1 << std::endl;
15 }
```

`mymax` 함수는 문자열의 처음 글자인 `a`와 `x`를 비교하여 `x`가 더 뒤에 있으므로 `s2`를 반환하게 된다. **만약 문자열의 순서가 아니라 문자열의 개수를 `mymax`를 통해 비교하고 싶다면 어떻게 해야 할까?**

이와 같이 알고리즘 함수가 사용하는 "정책(비교 방식)"을 변경하고 싶은 경우 다음과 같은 방법을 사용 할 수 있다.

- 이항 조건자(binary predicator) 사용 (e.g., c++ stl)

```
1 std::sort(v.begin(), v.end(), [](auto &a, auto &b) { return a.size() < b.size(); });
```

- 단항 조건자(unary predicator) 사용 (e.g., python)

```
1 sorted(str_list, key = lambda x : len(x));
```

- 단항, 다항 조건자를 모두 사용 (e.g., c++ 20 range 알고리즘의 원리). 멤버 함수 포인터, 멤버 데이터 포인터, `std::invoke`를 모두 활용한다.

```
1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return proj(obj1) < proj(obj2) ? obj2 : obj1; // proj()를 통해 비교하고 결과값은 원래 값을 반환한다.
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); });
11    std::cout << ret1 << std::endl;
```

`mymax`의 3번째 인자로 단항 조건자를 전달하면 비교 시 조건자의 결과를 비교한다. 이는 c++20에서 **Projection**이라는 기술로 불린다.

만약 단항 조건자 대신 멤버 함수 포인터를 전달할 수는 없을까?

```

1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return proj(obj1) < proj(obj2) ? obj2 : obj1;
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); });
11    auto ret2 = mymax(s1, s2, &std::string::size); // error. 멤버 함수 포인터이므로 proj(obj)에서
12        // 에러가 발생한다.
13 }
```

`proj(obj)`는 일반 함수라면 `ok`이지만 멤버 함수라면 에러가 발생한다. 이 때, `std::invoke`를 사용하면 일반 함수와 멤버 함수 모두 커버할 수 있다.

```

1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1; // 일반 함수, 멤버 함수
4         // 모두 커버 가능
5 }
6
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); }); // ok
11    auto ret2 = mymax(s1, s2, &std::string::size); // ok.
12 }
```

Projection은 생략될 수 있어야 한다. 이를 위해서는 `Proj`의 디폴트 값이 있어야 한다. `std::identity`는 전달 받은 인자를 어떤 변화 없이 참조값을 반환하는 함수 객체이다. 이를 활용하여 디폴트 값을 정의한다.

```

1 template<class T, class Proj = std::identity> // std::identity로 디폴트 값 설정
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz"
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); }); // ok
11    auto ret2 = mymax(s1, s2, &std::string::size); // ok.
12    auto ret3 = mymax(s1, s2); // ok
13 }
```

`std::identity`은 c++20에 처음 등장하였으므로 구현 코드는 다음과 같이 되어 있다. `<functional>` 헤더 파일이 필요하다.

```

1 struct identity {
2     template<class _Ty>
3     [[nodiscard]] constexpr
4     _Ty&& operator() (_Ty&& arg) const noexcept {
5         return std::forward<_Ty>(arg);
6     }
7     using is_transparent = int;
8 }
```

다음으로 mymax를 통해 임의의 객체 Point를 비교해보자.

```
1 template<class T, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4     // (obj1.*proj) < (obj2.*proj)
5 }
6
7 struct Point {
8     int x,y;
9 }
10
11 int main() {
12     Point p1 = {2,0};
13     Point p2 = {1,1};
14
15     auto ret = mymax(p1, p2, &Point::y); // 이렇게 비교할 수는 없을까?
16     cout << ret.x << ", " << ret.y << endl;
17 }
```

std::invoke는 멤버 변수의 포인터도 커버할 수 있다. 따라서 위 코드는 아무 수정 없이 정상적으로 동작한다. 정리하면 mymax는 다음과 같이 4가지 사용법이 존재한다.

```
1 string s1 = "abcd";
2 string s2 = "xyz";
3
4 auto ret1 = mymax(s1, s2); // (1)
5 auto ret2 = mymax(s1, s2, [] (auto &a) { return a.size(); }); // (2)
6 auto ret3 = mymax(s1, s2, &std::string::size); // (3)
7
8 Point p1 = {0,0};
9 Point p2 = {1,1};
10
11 auto ret4 = mymax(p1, p2, &Point::y); // (4)
```

2.7.1 Add compare operator

앞서 살펴본 mymax 함수는 std::invoke를 사용하여 여러 케이스에 대해 사용 가능한 훌륭한 함수였다.

```
1 template<class T, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4 }
```

mymax 함수는 < 연산을 기본으로 하고 있다. 이를 유저가 원하는 임의의 비교 연산자를 사용할 수는 없을까? 세번째 인자에 Comp 연산자를 추가함으로서 이를 가능하게 할 수 있다.

```
1 template<class T, class Comp = std::less<void>, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Comp comp = {}, Proj proj = {}) {
3     return std::invoke(comp, std::invoke(proj, obj1), std::invoke(proj, obj2)) ? obj2 : obj1;
4     // comp가 멤버 함수일 때는 comp(a,b)가 불가능하기 때문에 std::invoke(comp, a, b)로 작성해준다.
5 }
6
7 int main(){
8     std::string s1 = "abcd";
9     std::string s2 = "xyz";
10
11     auto ret1 = mymax(s1, s2);
12     auto ret2 = mymax(s1, s2, std::greater{});
13     auto ret2 = mymax(s1, s2, {}, &std::string::size); // 세번째 {}는 디폴트 연산자를 사용하라는 의미
14     auto ret2 = mymax(s1, s2, std::greater{}, &std::string::size);
15 }
```

왜 템플릿의 디폴트 인자를 std::less<T>가 아닌 std::less<void>로 했을까? 이는 callable section을 공부하고 오면 답을 알 수 있다.

2.7.2 c++20 ranges algorithm

지금까지 구현한 `mymax`는 c++20에서 도입된 range algorithm 기반의 `std::ranges::max` 함수와 거의 동일한 구현 코드를 갖는다. 하지만 이번 섹션에서 만든 `mymax`은 함수 템플릿이지만 `std::ranges::max`은 함수 객체(템플릿)임에 유의한다.

```
1 template<class T, class Comp = std::less<void>, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Comp comp = {}, Proj proj = {}) {
3     return std::invoke(comp, std::invoke(proj, obj1), std::invoke(proj, obj2)) ? obj2 : obj1;
4 }
5
6 int main(){
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2);
11    auto ret2 = mymax(s1, s2, std::greater{}, &std::string::size);
12    auto ret3 = std::ranges::max(s1, s2, std::ranges::greater, &std::string::size); // ret2와 동일한
13                                결과 출력
}
```

c++20의 range algorithm은 알고리즘의 "비교 정책"을 교체할 수 있으며 "Projection"을 전달할 수 있다. 이 때 `std::invoke`를 통해 구현하여 멤버 함수 포인터와 멤버 데이터 포인터 모두 사용이 가능하다. 또한 반복자 구간이 아닌 컨테이너를 전달받기 때문에 다음과 같이 조금 더 편하게 코딩이 가능하다.

```
1 std::vector<std::string> v = {"hello", "a", "xxx", "zz"};
2
3 std::sort(v.begin(), v.end()); // 매 번 begin, end를 적어줘야 하므로 불편함
4
5 // c++20 ranges
6 std::ranges::sort(v); // 컨테이너만 넘겨주면 자동으로 정렬. 디폴트로 알파벳 순서대로 정렬
7 std::ranges::sort(v, std::ranges::greater{}, &std::string::size); // 비교 정책과 Projection 모두
8 전달 가능
```

2.8 Size of member function pointer

이번 섹션에서는 멤버 함수 포인터의 크기를 알아본다. 이를 알기 위해서는 우선 다중상속을 정확히 이해하고 있어야 한다.

```
1 struct A {int x;};
2 struct B {int y;};
3 struct C : public A, public B { // 다중 상속
4     int z;
5 };
6
7 int main(){
8     C cc;
9     cout << &cc << endl; // 1000
10
11    A* pA = &cc;
12    B* pB = &cc;
13    cout << pA << endl; // 1000
14    cout << pB << endl; // 1004. B클래스는 두번째로 상속을 받고 있으므로 A 클래스의 멤버 변수가 차지하는 4
15                                바이트를 견너뛰어 1004의 값을 갖는다
}
```

`static_cast`를 해도 똑같이 1004의 값이 나온다. `reinterpret_cast`를 사용해야 1000의 값이 나온다. `reinterpret_cast`은 0x1000 위치의 메모리를 B타입처럼 사용하겠다라는 의미이다. 메모리를 다르게(다른 타입으로) 사용하겠다는 의미이므로 사용에 주의해야 한다.

```
1 B* pB1 = static_cast<B*>(&cc); // 1004
2 B* pB2 = reinterpret_cast<B*>(&cc) // 1000
```

다음으로 멤버 함수가 있는 예제를 보자.

```

1  struct A{
2      int x;
3      void fa() { std::cout << this << std::endl; }
4  };
5  struct B{
6      int y;
7      void fb() { std::cout << this << std::endl; }
8  };
9  struct C : struct A, struct B{
10     int z;
11     void fc() { std::cout << this << std::endl; }
12 };
13
14 int main() {
15     C cc;
16     cc.fc(); // 0x1000
17     cc.fa(); // 0x1000
18     cc.fb(); // 0x1004. 이전 코드와 동일하게 B에 대한 상속이 두번째로 받았으므로 메모리 공간에 4바이트만큼
19         뒤로 간 1004가 출력된다.
}

```

다음과 같이 함수 포인터가 있다고 하자.

```

1 void (C::*f)();
2
3 f = &C::fa;
4 (cc::*f)(); // 1000
5
6 f = &C::fb;
7 (cc::*f)(); // 1004. 정상적으로 함수 포인터가 동작한다.

```

함수 포인터는 런타임 시점에서 `f(&cc)`와 같이 호출된다. 하지만 `f = &C::fb`를 가리키는 경우 1004가 정상적으로 호출되려면 `f(&cc + sizeof(A))`가 호출되어야 한다. **컴파일 타임에선 f가 fa를 가리킬지 fb를 가리킬지는 알 수 없는데 어떻게 정상적으로 동작하는 것일까?**

멤버 함수 포인터는 항상 4byte(32bit), 8byte(64bit)인 것은 아니다. 코드에 따라 멤버 함수 포인터의 크기가 달라지는데 **다중 상속의 경우 함수 주소와 this offset을 같이 보관하여 8byte(32bit), 16byte(64bit)의 크기를 가진다.**

```

1 void(C::*f)(); // 함수 주소 + this offset을 함께 보관 8byte(32bit), 16bit(64bit)
2
3 f = &C::fa; // fa주소, 0이 보관
4 f = &C::fb; // fb주소, sizeof(A) 보관

```

위와 같은 동작이 c++ 표준 동작은 아니다. 컴파일러마다 원리가 다를 수 있으나 대부분의 컴파일러가 위와 같은 방법을 통해 멤버 함수 포인터를 관리한다.

Tip

`void*`가 모든 주소를 담는다고 알려져 있지만 엄밀하게 보면 {주소, this offset}을 보관하는 **멤버 함수의 포인터는 담을 수 없다.** 또한 멤버 데이터의 포인터도 진짜 메모리 주소가 아닌 상대적인 offset 정보만 가지고 있기 때문에 **멤버 데이터 포인터 또한 담을 수 없다.**

```

1 struct myostream{
2     myostream& operator<<(int n) { printf("int : %d\n", n); return *this; }
3     myostream& operator<<(double d) { printf("double : %f\n", d); return *this; }
4     myostream& operator<<(bool b) { printf("bool %d\n", b); return *this; }
5     myostream& operator<<(void* p) { printf("void*: %p\n", p); return *this; }
6 }
7 myostream mycout;
8
9 int main() {
10     int n=10;
11     double d=3.4;

```

```

12     int Point::*p = &Point::y;
13
14     mycout << n;      // int
15     mycout << d;      // double
16     mycout << &n; // void*
17     mycout << &d; // void*
18     mycout << p; // 1. 1이 출력되는 이유는 void*로 변환될 수 없는 멤버 변수 포인터가 bool로는 암시적
19         변환이 되기 때문에 4 -> 1(true)가 되어 1이 출력되게 되는 것이다.
}

```

따라서 멤버 함수의 포인터를 출력해보고 싶을 때는 cout보단 printf를 써서 출력하는 것이 좋다.

2.9 new, delete and placement new

이번 섹션에서는 `new`, `delete` 키워드에 대해서 자세히 살펴본다.

```

1 class Point{
2     int x,y;
3     public:
4     Point(int a, int b) : x{a}, y{b} { cout << "Point(int,int)" << endl; }
5     ~Point() { cout << "~Point()" << endl; }
6 }
7
8 int main(){
9     Point* p1 = new Point(1,2); // 메모리 할당, 생성자 호출
10    delete p1; // 소멸자 호출, 메모리 해제
11 }

```

c++에서 `new` 키워드를 사용하면 다음과 같은 메모리 할당, 생성자 호출 코드가 자동으로 수행된다.

```

1 Point* p1 = new Point(1,2);
2
3 void* p = operator new(sizeof(Point)); // 1. 메모리 할당
4 Point* p1 = new(p) Point(1,2); // 2. 생성자 호출. new(p)를 placement new라고 부른다.

```

`delete`를 사용하면 소멸자가 호출되고 메모리가 해제되는 코드가 자동으로 수행된다.

```

1 delete p1;
2
3 p1->~Point(); // 1. 소멸자 호출
4 operator delete(p1); // 2. 메모리 해제

```

만약 생성자 호출 없이 메모리만 할당, 해제하고 싶은 경우 다음과 같이 쓰면 된다. c언어에서 `malloc`과 `free`한 것과 동일하다.

```

1 void* p = operator new(sizeof(Point)); // Point 구조체는 x,y로 인해 8바이트 할당
2
3 operator delete(p); // 메모리 해제

```

`operator new`, `delete` 함수는 c++20 기준으로 각각 22개, 30개의 다른 버전이 있다. 두 함수는 `<new>` 헤더를 필요로 하며 std namespace가 아닌 global namespace에 존재한다.

```

1 [[nodiscard]] void* operator new(std::size_t);
2 void operator delete (void* ptr) noexcept;

```

앞서 메모리를 할당한 다음 생성자를 호출하려면 아래와 같이 하면 된다.

```

1 void *p1 = operator new(sizeof(Point));
2 Point *p2 = new(p1) Point(1,2); // 생성자 호출

```

`new(p1)`은 **placement new**라고 불리며 메모리 할당없이 이미 할당된 메모리에 생성자를 명시적으로 호출하기 위한 `new`이다. c++20에서는 `new(p) Point(1,2)` 대신 `std::construct_at(p, 1, 2)`를 통해 생성자를 명시적으로 호출할 수 있다. 유사하게 `std::destroy_at(p)`를 통해 소멸자를 호출할 수 있다.

`void*`가 아닌 정확한 클래스 타입으로 메모리를 할당받기 위해서는 다음과 같이 작성한다.

Tip

- `new Point(1,2);` // 새로운 메모리를 할당하고 객체 생성
- `new(p) Point(1,2);` // 이미 할당된 메모리(p)에 객체 생성 (=placement new)
- `std::construct_at(p, 1, 2);` // 위와 동일. c++20, <memory>
- `std::destroy_at(p);` // 명시적 소멸자 호출

```
1 Point* p3 = static_cast<Point*>(operator new(sizeof(Point)));
2 std::construct_at(p3, 1, 2); // 생성자 호출
```

2.9.1 Using placement new

메모리 할당과 생성자 호출을 분리해야 하는 이유가 있을까?

```
1 class Point{
2     int x,y;
3     public:
4     Point(int a, int b) :x{a}, y{b} {} // 디폴트 생성자가 없음에 주목!
5     ~Point() {}
6 };
7
8 int main(){
9     Point* p1 = new Point(0,0); // Point 객체 한개를 힙에 생성하고 싶은 경우
10
11    Point* p2 =? // 만약 Point 객체 3개를 연속적인 형태(배열 형태)로 생성하고 싶은 경우 어떻게 해야 할까?
12 }
```

p2를 `new Point[3]`와 같이 생성하면 Point 타입에는 반드시 디폴트 생성자가 있어야 한다. 하지만 **디폴트 생성자 없이도 3개를 연속적인 배열 형태로 만들고 싶은 경우도 발생할 수 있다.**

c++11에는 다음과 같이 초기화할 수 있다. `{0,0}`은 인자 2개짜리 생성자를 호출하기 때문에 예러없이 잘 빌드된다.

```
1 Point* p2 = new Point[3]{{0,0}, {0,0}, {0,0}};
```

하지만 3개가 아니라 30개, 100개가 넘어가는 경우는 어떻게 해야 할까? 이럴 때는 **메모리 할당과 생성자 호출을 분리하면 훨씬 편하게 생성할 수 있다.**

```
1 Point* p2 = static_cast<Point*>(operator new(sizeof(Point)*3));
2
3 for(int i=0; i<3; i++){
4     new(&p2[i]) Point(0,0); // c++20 이전 표기법
5     std::construct_at(&p2[i], 0,0); // c++20 이후 표기법
6 }
7
8 for(int i=0; i<3; i++){
9     p2[i].~Point(); // c++20 이전 표기법
10    std::destroy_at(&p2[i]); // c++20 이후 표기법
11 }
```

2.9.2 vector and placement new

```
1 int main(){
2     vector<int> v(10);
3
4     v.resize(7); // size의 크기를 줄이고 메모리 할당 크기는 capacity 변수에 저장한다.
5     cout << v.size() << endl; // 7
6     cout << v.capacity() << endl; // 10
```

```

7     v.resize(8);
8     cout << v.size() << endl; // 8
9     cout << v.capacity() << endl; // 10
10 }

```

벡터가 임의의 클래스 X를 담고 있다고 해보자.

```

1 struct X{
2     X() { cout << "X() get resource" << endl; }
3     ~X() { cout << "~X() release resource" << endl; }
4 }
5
6 int main(){
7     vector<X> v(10);
8
9     v.resize(7); // 크기가 줄어드는 경우 메모리는 제거하지 않더라도 소멸자는 불러야하지 않을까?
10
11    v.resize(8); // 8번째 메모리에 생성자가 호출되어야 하지 않을까?
12 }

```

사용자 정의 타입을 vector에 담으면 **메모리의 할당, 해지가 없는데도 불구하고 생성자, 소멸자의 호출을 정말 많이 호출**한다. 따라서 메모리 할당, 생성자 호출에 대한 개념의 정확한 이해가 필요하다.

2.10 Trivial constructor

2.10.1 Trivial default constructor

생성자가 trivial하다는 말은 컴파일러가 자동으로 생성해주면서 동시에 아무 일도 하지 않을 때를 말한다

2.10.2 Trivial copy constructor

복사 생성자가 trivial하다는 말은 멤버변수 값을 복사하는 것 이외에 아무 일도 하지 않을 때를 말한다. 복사 생성자가 trivial하다면 배열 전체를 `memcpy`, `memmove` 등으로 복사하는 것이 빠르다! 복사 생성자가 trivial하지 않다면 배열의 모든 요소에 대해 하나씩 “복사 생성자”를 호출해서 생성자를 호출해야 한다

```

1 struct Point {
2     int x=0;
3     int y=0;
4 };
5
6 template<class T>
7 void constexpr copy_type(T* dst, T* src, std::size_t sz) {
8     if(std::is_trivially_copy_constructible_v<T>) {
9         std::cout << "using memcpy" << std::endl;
10        memcpy(dst, src, sizeof(T)*sz);
11    }
12    else {
13        std::cout << "using copy ctor" << std::endl;
14        while(sz--){
15            new(dst) T(*src);
16            --dst, --src;
17        }
18    }
19 }
20
21 int main(){
22     Point arr1[5];
23     Point arr2[5];
24     copy_type(arr1, arr2, 5);
25 }

```

위 코드에서 Point 클래스는 `int x,y`와 같이 간단한 멤버변수만 존재하므로 trivial copy constructor이다. 하지만 `virtual void foo(){}` 같이 가상함수를 사용하거나 `string s;`와 같이 복사하는 클래스를 사용하게 되면 trivial하지 않게 된다. 이런 경우에는 `placement new` 또는 `std::construct_at`을 사용해야 한다.

2.11 Type deduction

컴파일 타입에 타입이 결정되는 `auto` 키워드에 대해 살펴보자

```
1 int main(){
2     int n=10;
3     const int c =10;
4
5     auto a1 = n; // int a1=n;
6     auto a2 = c; // (1) const int a2 = c; --> no.
7     // (2) int a2 = c; --> ok.
8 }
```

type deduction(타입 추론)이 발생하는 키워드는 다음과 같다: `template`, `auto`, `decltype`

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << typeid(T).name() << std::endl;
4 }
5 int main(){
6     int n=10;
7     foo(n); // T=int
8     foo<const int&>(n); // T=const int&. But typeid(T).name() keep printing output 'int'
9 }
```

`typeid(T).name()`은 타입 이름만 추론할 뿐 `const`, `volatile`, `reference` 정보가 출력되지 않는다.

1. 이럴 때는 의도적으로 에러를 발생시켜서 정확한 타입을 에러 메시지를 통해 알 수 있다.
2. 또는 `boost::type_index` 라이브러리에 `type_id_with_cvr<T>().pretty_name()`을 사용하면 됨
3. 컴파일러가 제공하는 매크로를 사용하면 된다.
 - `__FUNCTION__`: 함수의 이름만 보여주므로 타입 추론에는 사용하지 않음
 - `__PRETTY_FUNCTION__`: g++, clang에서는 함수 이름과 타입이 나옴
 - `__FUNCSIG__`: cl.exe 버전

```
1 std::cout << __FUNCTION__ << std::endl;
2 std::cout << __PRETTY_FUNCTION__ << std::endl; // g++, clang
3 std::cout << __FUNCSIG__ << std::endl; // cl.exe
```

T가 값인 경우를 살펴보자

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     while(--arg>0) {}
4 }
5
6 int main(){
7     int n=10;
8     int& r = n;
9     const int c = 10;
10    const int& cr = c;
11    foo(n); // T=int
12    foo(r); // T=int& 일 것 같지만 T=int
13    foo(c); // T=const int 일 것 같지만 T=int
14    foo(cr); // T=const int& 일 것 같지만 T=int
15 }
```

T 인자를 값으로 받을 때는 복사본 객체가 만들어져서 “`const`, `volatile`, `reference`” 속성을 제거하고 값만 받는다. 헷갈리는 것 중 하나가 값으로 받을 때는 인자의 `const` 속성은 제거되고 “인자가 가리키는 곳의 `const` 속성은 유지”한다. 무슨 이야기인지 살펴보자

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
```

```

4     }
5     int main(){
6         const char* const s = "hello";
7         foo(s); // 포인터의 const 속성은 제거되지만 가리키는 곳 "hello"의 const 속성은 유지됨!
8         // const char* arg = "hello"가 됨!!
9     }

```

T 인자를 참조로 받을 때는 다음과 같다.

```

1 #include <iostream>
2 template<class T> void foo(T& arg){
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
4 }
5
6 int main(){
7     int n = 10;           // T=int.      arg=int&
8     int& r = n;          // T=const int. arg=const int& (const 속성은 유지!)
9     const int c = 10;    // T=int        arg=int&. (T에서 reference 속성은 제거!)
10    const int& cr = c; // T=const int   arg=const int& (const 속성은 유지!)
11 }

```

주의해야 할 점은 T의 타입과 arg의 타입은 다르다는 것이다 (`T& arg`이기 때문!). 함수 인자의 “reference를 제거하고 T의 타입을 결정한다”. 인자가 가진 “`const, volatile` 속성은 유지한다.”. 마지막으로 T에 배열이 전달된 경우를 살펴보자

```

1 template<class T> void foo(T arg){
2     std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5     std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9     int x[3] = {1,2,3};
10    foo(x); // T=int* 타입으로 받는다
11    goo(x); // T=int[3] 타입으로 받는다. arg는 int() [3] 타입으로 받는다
12 }

```

`T& arg`로 배열을 받는 경우 `int (&arg)[3] = x;` 처럼 받는게 되어서 배열의 reference가 된다. 따라서 아래와 같이 `goo()`를 사용하면 에러가 발생한다.

```

1 template<class T> void foo(T arg){
2     std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5     std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9     foo("orange", "apple"); // ok
10    goo("orange", "apple"); // error
11 }

```

`foo`와 `goo` 둘 다 `const char` 형식으로 받지만 포인터는 개수에 제한이 없으므로 ok인 반면에 reference는 `const char[7]`, `const char[6]`은 다른 reference이기 때문에 같은 T를 받는 상황에서 에러가 발생한다!
`foo(const char[7], const char[6])`, `goo(const char[7], const char[6])`

2.11.1 Auto type deduction

`template`은 함수 인자로 추론하는 반면 `auto`는 우변의 표현식으로 타입을 추론한다. `template`을 ‘`T arg = parameter`’처럼 추론한다고 볼 수 있으므로 사실 ‘`auto a = expression`’과 동일한 형태로 추론한다!

```

1 int main(){
2     int n=10;
3     int& r = n;

```

```

4     const int c = 10;
5     const int& cr = c;
6
7     auto a1 = n; // auto=int
8     auto a2 = r; // auto=int
9     auto a3 = c; // auto=int
10    auto a4 = cr; // auto=int
11
12    auto& a5 = n; // auto=int. a5=int&
13    auto& a6 = r; // auto=int. a6=int&
14    auto& a7 = c; // auto=const int. a7=const int&
15    auto& a8 = cr; // auto=const int. a8=const int&
16
17    int x[3] = {1,2,3};
18    auto a = x; // auto=int*
19    auto& b = x; // auto=int[3]. b=int(&) [3]
20 }
```

위와 같이 template에서 본 규칙들이 그대로 적용! 아래와 같이 조금 더 까다로운 경우를 보자.

```

1 int main(){
2     auto a1 = 1;
3     auto a2 = {1};
4     auto a3{1};
5
6     std::cout << typeid(a1).name() << std::endl; // auto=int
7     std::cout << typeid(a2).name() << std::endl; // auto=initialized_list
8     std::cout << typeid(a3).name() << std::endl; // auto=int
9     std::vector<int> v1(10,0);
10    std::vector<bool> v2(10,false);
11
12    auto a4 = v1[0];
13    auto a5 = v2[0];
14
15    std::cout << typeid(a4).name() << std::endl; // auto=int
16    std::cout << typeid(a5).name() << std::endl; // auto=temporary proxy 객체
17 }
```

배열은 `auto a= {1}`라고 하면 배열 타입으로 추론된다. `bool` 타입은 최적화 과정에서 specialization되어 있다. 따라서 `bool`은 [] 연산자가 bool로 변환 가능한 `temporary proxy`으로 추론한다!

2.11.2 Array Name

배열의 이름은 배열의 1번째 요소의 주소로 암시적 형변환 된다.

```

1 int main(){
2     int x[3] = {1,2,3};
3
4     int *p0[3] = &x; // error. 연산자 우선 순위에 따라 p[3], *p 순으로 추론된다.
5     int (*p1)[3] = &x; // ok. (*p)로 감싸줘야 x[3] 배열에 대한 제대로된 포인터가 된다.
6     int *p2 = x; // ok. &x[0]
7
8     printf("%p, %p\n", p1, p1+1); // 배열 자체를 가리키므로 +1을 하면 12바이트만큼 증가한다.
9     printf("%p, %p\n", p2, p2+1); // 배열의 첫번째 원소를 가리키므로 +1을 하면 4바이트만큼 증가한다.
10
11    (*p1)[0] = 10;
12    *p2 = 10;
13 }
```

위 코드에서 보면 `p1`, `p2`가 가리키는 주소는 동일하지만 포인터 타입이 다르므로 주소를 해석하는 방식이 다르다. 다음과 같이 배열을 함수 인자를 받는 경우에 대해 알아보자

```

1 void f1(int p[3]) {
2     printf("%d\n", sizeof(p)); // 마치 3개 배열을 입력으로 받는 듯 보이지만 int *p와 동일한 포인터를 받고
3     // 있다. 즉, 12가 아닌 8(64bit)이 나온다
4 }
```

```

4
5     int main(){
6         int x[3] = {1,2,3};
7         f1(x);
8     }

```

함수 인자로 배열을 받을 때는 `int *p`와 같이 포인터 타입으로 받거나 `int p[]`와 같이 배열 타입으로 받는다. 이 때 컴파일러에 의해 둘 다 **포인터**로 변환한다. (`int *p`).

`int p[3]`과 같이 배열의 크기도 지정해줄 수 있는데 이 또한 **포인터**로 컴파일러가 변환한다(`int *p`). 헷갈리기 쉬운 문법이니 주의한다. 마지막으로 다차원 배열의 포인터에 대해 알아보자

```

1     void foo(int (*p)[2]) {
2         p[0][0] = 100;
3     }
4     int main(){
5         int y[3][2] = {1,2,3,4,5,6};
6
7         int (*p3)[3][2] = &y; // 배열의 변수와 정확히 동일한 형태를 유지하고 (*p)로 감싸주면 배열 포인터가
8         된다.
9         int (*p4)[2] = y;    // 배열의 첫번째 원소는 2차원 배열이므로 (=1,2) int (*p4)[2]와 같이 선언해줘야
10        한다.
11
12     foo(y);
13 }

```

2.12 Rvalue & forwarding & reference

2.12.1 Lvalue vs Rvalue

- lvalue: 등호의 왼쪽에 올 수 있는 표현식
- rvalue: 등호의 왼쪽에 올 수 없는 표현식

이외에도 c++에서는 다음과 같은 추가적인 구분 방법이 존재한다.

```

1     x=10;
2     int f1() { return x; }
3     int& f2() { return x; }
4
5     int main(){
6         int v1=0, v2=0;
7
8         v1=10; // ok. v1 : lvalue
9         10=v1; // error. 10 : rvalue
10        v2=v1;
11        int *p1 = &v1; // ok
12        int *p2 = &10; // error
13
14        f1() = 20; // error
15        f2() = 20; // ok
16        const int c = 10; // 상수도 lvalue의 특징을 가지고 있다 (이름, 주소)
17        c = 20; // error
18        "aa"[0] = 'x'; // error. lvalue 문제가 아니라
19        // const char[3]이므로 에러 발생!
20    }

```

이름, 주소가 있으면 lvalue, 없으면 rvalue. 참조를 반환하면 lvalue, 값을 반환하면 rvalue라고 볼 수 있다. 다음과 같은 의문이 들 수 있다.

1. **모든 상수는 rvalue인가?** : 상수는 immutable lvalue로 취급한다.
2. **모든 rvalue는 상수인가?** : `Point(1,2).set(10,20)` 같이 temporary 객체도 멤버 변수를 호출할 수 있으므로 상수가 아니다

lvalue, rvalue에 대한 혼란 오해 중 하나가 객체, 변수에 부여되는 속성으로 오해한다. **하지만 이는 표현식 (expression)에 부여되는 속성이다!** 표현식이란 “하나의 값”을 만들어내는 코드 집합을 말한다!

```

1 int main(){
2     int n=3;
3
4     n=10;      // ok
5     n+2 = 10; // error. n+2=5인데 이는 값이므로 rvalue!
6     n+2*3 = 10; // error.
7
8     (n=20) = 10; // ok. 표현식 ok
9
10    ++n = 10; // ok
11    n++ = 10; // error. n=3-->4로 바뀌는데 이는 값이므로 error
12 }
```

어떤 값이 lvalue인지 rvalue인지 조사하려면 **decltype**을 사용하면 된다!

```

1 int main(){
2     int n = 10;
3
4     if(std::is_lvalue_reference_v<decltype(n++)>) // n++: rvalue, ++n: lvalue
5         std::cout << "lvalue" << std::endl;
6     else
7         std::cout << "rvalue" << std::endl;
8
9     if(std::is_lvalue_reference_v<decltype((n))>) // 그냥 n을 넣으면 rvalue로 잘못나온다. (n)을 통해
10    괄호로 감싸줘야 표현식으로 인식해서 lvalue로 정상 인식한다!
11    std::cout << "lvalue" << std::endl;
12 else
13     std::cout << "rvalue" << std::endl;
14 }
```

다음과 같이 매크로를 만들어 놓으면 편하게 구분할 수 있다.

```

1 #define value_category(...)
2 if( std::is_lvalue_reference_v<decltype((__VA_ARGS__))> )
3     std::cout << "lvalue" << std::endl;
4 else if( std::is_rvalue_reference_v<decltype((__VA_ARGS__))> )
5     std::cout << "rvalue(xvalue)" << std::endl;
6 else
7     std::cout << "rvalue(prvalue)" << std::endl;
8
9 int main(){
10     int n=10;
11
12     value_category(n);
13     value_category(n+2);
14     value_category(n++);
15     value_category(++n);
16 }
```

2.12.2 Reference & Overloading

참조자(reference)의 규칙에 대해 알아보자

```

1 int main(){
2     int n=3;
3
4     int& r1 = n; // ok
5     int& r2 = 3; // error
6     const int& r3 = n; // ok
7     const int& r4 = 3; // ok (하지만 상수성이 추가됨)
8
9     int&& r5 = n; // error
10    int&& r6 = 3; // ok. (상수 성질 없음! rvalue reference라고 부름)
11 }
```

임의의 숫자를 가리키고 싶을 땐 `const int& r4 = 3`과 같이 가리켜야 했으나 c++11 이후 rvalue-reference라는 문법이 등장하면서 `int&& r6=3`과 같이 가리킬 수 있게 되었음.

기존 `int& r1`을 lvalue-reference라고 부르며 `int&& r6=3`을 rvalue-reference라고 부름! 그렇다면 왜 상수성 없이 rvalue를 가리키는 것이 중요할까? 이는 move semantics와 perfect forwarding을 위해서 필요하다. 자세한 내용은 추후 다룰 예정이다. 다음으로 rvalue, lvalue의 함수 오버로딩에 대해 알아보자

```
1 class X{};  
2  
3 // void foo(X x) { std::cout << "X" << std::endl }. // 값 타입과 참조 타입은 서로 오버로딩될 수 없다!  
4 void foo(X& x) { std::cout << "X&" << std::endl } // 1  
5 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2  
6 void foo(X&& x) { std::cout << "X&&" << std::endl } // 3  
7 //void foo(const X&& x) { std::cout << "const X&&" << std::endl }  
8  
9 int main(){  
10     X x;  
11     foo(x); // lvalue. 어느 곳에 오버로딩 해야할지 몰라서 에러 발생  
12     // 값 타입을 주석처리하면 X x에 오버로딩 됨. (1, 2) 순서로 오버로딩  
13     foo(X()); // rvalue. 어느 곳에 오버로딩 해야할지 몰라서 에러 발생  
14     // 값 타입을 주석처리하면 X x에 오버로딩 됨. (3, 2) 순서로 오버로딩  
15 }
```

`foo` 함수의 마지막 표기법 `const X&& x`는 문법적으로는 가능하지만 사용하지 않는다. Move semantic을 통해 대체할 수 있기 때문이다. 다음은 주로 헷갈리는 문법에 대해 알아보자.

```
1 class X{};  
2  
3 void foo(X& x) { std::cout << "X&" << std::endl } // 1  
4 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2  
5 void foo(X&& x) { std::cout << "X&&" << std::endl } // 3  
6  
7 int main(){  
8     foo( X() ); // 3  
9  
10    X&& rx = X();  
11    foo(rx); // 3번이 호출될 것 같지만 1번이 호출됨! lvalue로 인식하기 때문  
12    foo(static_cast<X&&>(rx)); // 3  
13 }
```

`X&& rx = X()`에서 `rx`는 rvalue라고 생각할 수 있으나 이름이 있으므로 lvalue로 취급된다. 따라서 함수를 호출하면 1번이 호출된다!

따라서 3번을 호출하고 싶으면 `static_cast<X&&>()`을 사용하여 형변환을 하면 되는데 자세히 보면 같은 타입을 왜 변환해야 하는가? 하는 의문이 생길 수 있다. 이는 특수한 케이스로써 c++ 문법에서 타입 캐스팅이 아닌 value를 변환하는 캐스팅으로 기재되어 있다.

2.12.3 Reference collapsing

```
1 int main(){  
2     int n=3;  
3     int& lr = n; // lvalue reference  
4     int&& rr = 3; // rvalue reference  
5  
6     int& &ref2ref = lr; // error! 명시적으로 레퍼런스를 가리키는 레퍼런스는 코딩 불가  
7  
8     decltype(lr)& r1 = ? // int& & ==> int&  
9     decltype(lr)&& r2 = ? // int& && ==> int&  
10    decltype(rr)& r3 = ? // int&& & ==> int&  
11    decltype(rr)&& r4 = ? // int&& && ==> int&& 모두 두개씩 있을 때만 rvalue refernce로 인식!  
12 }
```

레퍼런스를 가리키는 레퍼런스는 명시적으로 코딩이 불가능하다. 하지만 `decltype()`을 통해 타입을 추론하게 되면 가능하다.

`decltype(&&)&&` 인 경우에만 `int&&` 타입으로 추론하고 나머지 경우는 `int&`로 추론한다. 이런 추론 규칙을 **reference collapsing**이라고 한다! reference collapsing은 **typedef, using, decltype, template** 4가지 경우에 적용된다.

```
1  template<typename T> void foo(T&& arg) { }
2
3  int main(){
4      int n=3;
5
6      typedef int& lref;
7      lref&& r1 = n;      // int& && ==> int&
8
9      using rref = int&&;
10     rref&& r2 = 10;    // int&& && ==> int&&
11
12     decltype(r2)&& r3 = 10; // int&& && ==> int&&
13
14     foo<int&>(n);      // foo(int& && arg)
15     // foo(int& arg) 함수 생성!
16 }
```

2.12.4 Forwarding reference

```
1  void f1(int& arg) {}
2  void f2(int&& arg) {}
3  template<typename T> void f3(T& arg) {}
4
5  int main(){
6      int n=3;
7
8      f1(n); // ok
9      f1(0); // error
10
11     f2(n); // error
12     f2(0); // ok
13
14     f3(n); // ok. T는 어떤 값으로 받아도 lvalue reference이다.
15     f3(0); // error
16 }
```

`T&`는 어떤 타입으로 받아도 (`int&`, `int&&`) lvalue reference이다! 템플릿 케이스에 대해 보다 자세히 알아보자

```
1  template<typename T> void f3(T&& arg) \{\}
2
3  int main()\{
4      int n=3;
5
6      // 사용자가 명시적으로 타입을 추론한 경우
7      f3<int>(n);
8      f3<int\&>(n);
9      f3<int\&\&>(n); // 3가지 케이스 모두 int&로 추론되므로 lvalue만 올 수 있다!
10
11     f3(n); // ok. 사용자가 템플릿 인자를 전달하지 않으면 int로 추론한다
12     f3(0); // error
13 \}
```

다음으로 템플릿 인자에 `T&&`가 붙어있는 경우를 생각해보자.

```
1  template<typename T> void f4(T&& arg) {}
2
3  int main()\{
4      int n=3;
5
6      f4<int>(n); // int &&. ==> int&& rvalue reference
```

```

7     f4<int&>(n);    // int& && ==> int&. 이 케이스에서만 lvalue reference가 된다!
8     f4<int&&>(n);   // int&& && ==> int&& rvalue reference
9
10    f4(n); // ok. 컴파일러가 자동으로 int 타입으로 변환해서 넘겨준다 (int ==> int)
11    f4(0); // ok.
12 }
```

T&& 와 같이 작성하면 타입 추론 규칙에 따라 rvalue와 lvalue를 같이 전달할 수 있다! 헷갈리기 쉬운 것이 함수가 하나인데 두 인자를 받는것이 아니라 함수가 2개가 생성되서 각각 따로 받는다. 그리고 생성된 각 함수는 call-by-value가 아닌 call-by-reference를 사용해서 전달받는다. 이러한 T&& reference를 **forwarding(universal) reference**라고 부른다!

2.13 Move semantics

2.13.1 Move constructor

다음과 같이 얕은 복사(shallow copy)가 일어나는 경우를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10 }
11
12 int main(){
13     Person p1("john", 20);
14     Person p2 = p1; // 얕은 복사 발생!
15 }
```

복사 생성자를 명시적으로 정해주지 않으면 컴파일러가 모든 멤버 변수 함수를 얕은복사하게 된다.

```

1 Person(const Person& p) : age(p.age) {
2     name = new char[strlen(p.name)+1];
3     strcpy_s(name, strlen(p.name)+1, p.name);
4 }
```

클래스 내에 다음과 같은 복사 생성자를 정의해주면 더 이상 얕은 복사가 발생하지 않고 깊은 복사가 발생한다! 하지만 복사 생성자는 성능 이슈가 존재한다. 다음과 같이 임시 객체를 반환하는 함수를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10    Person(const Person& p) : age(p.age) {
11        name = new char[strlen(p.name)+1];
12        strcpy_s(name, strlen(p.name)+1, p.name);
13    }
14 }
15
16 Person foo() {
17     Person p("john", 20);
18     return p;
19 }
20
21 int main(){
22     Person ret = foo(); // 임시객체 반환하여 ret에 복사 생성자를 호출하여 복사해주고 바로 파괴됨!
```

}

`foo()` 함수는 값을 반환하므로 임시 객체가 생성되어 `ret`에 복사 생성자를 통해 깊은 복사를 일으키고 바로 파괴된다. `foo()`의 임시 객체를 파괴하지 않고 `ret`에 임시 객체의 주소를 그대로 가리킨 뒤 임시 객체의 메모리를 0으로 만드는 방법이 효율적이다!

```
1 Person(Person&& p) : name(p.name), age(p.age) {
2     p.name = nullptr;
3 }
```

Person 클래스 내부에 위와 같이 `move` 생성자를 만들면 rvalue만 받을 수 있고 임시 객체들은 해당 함수를 호출한다. 해당 코드가 rvalue를 처리하는 경우 기존의 복사 생성자보다 메모리 효율적이다!

2.13.2 std::move

```
1 class Object{
2     Object() = default;
3     Object(const Object& obj) { std::cout << "copy ctor" << std::endl; }
4     Object(Object&& obj) { std::cout << "move ctor" << std::endl; }
5 }
6
7 Object foo() {
8     Object obj;
9     return obj;
10 }
11
12 int main(){
13     Object obj1;
14     Object obj2 = obj1; // copy
15     Object obj3 = foo(); // move
16     Object obj4 = static_cast<Object&&>(obj1); // move
17     Object obj5 = std::move(obj2);           // move. 위 긴 변환문을 간단하게 move 함수를 호출하여 해결할
18     수 있다.
```

`std::move` 함수를 호출하면 `obj1, obj2` 같은 lvalue도 rvalue로 취급하여 `move` 생성자를 호출할 수 있다! `obj1, obj2`를 코드 내에서 더 이상 사용하지 않을 때 복사 생성자를 호출하기 보다 `move` 생성자를 호출하여 보다 효율적으로 값을 전달할 수 있다.

2.13.3 Move and noexcept

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::println("copy"); }
5     Object(Object&&) { std::println("move"); }
6 };
7
8 int main() {
9     std::vector<Object> v(3);
10    std::println("-----");
11    v.resize(5);           // copy, copy, copy 발생
12    std::println("-----");
13 }
```

`v(3)`로 처음 3개의 Object 자원을 확보한 후 `resize`를 통해 5개의 자원을 다시 확보한다고 5개 메모리를 새로 할당하고 기존 3개의 자원은 “복사(copy)”되어 새로운 메모리에 오게된다.

이렇게 `vector`의 베퍼를 새롭게 할당한 경우 결국 기존 베퍼를 제거하게 되므로 “복사(copy)”보다 “이동(move)”가 효율적이다! 하지만 위 코드를 실행하면 “복사(copy)”가 수행된다. 컴파일러가 기본적으로 복사를 수행하는 이유는 만약 이동을 하다가 예외가 발생하면 `vector`를 `resize` 이전 상태로 되돌릴 수 없다는 단점이 존재하기 때문이다.

따라서 이동을 사용하고 싶다면 되도록 예외가 발생하지 않도록 구현하고 `noexcept` 키워드를 붙여서 예외가 없음을

컴파일러에게 알려야 한다

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::println("copy"); }
5     Object(Object&&) noexcept { std::println("move"); } // 예외가 없음을 컴파일러에게 알림!
6 };
7
8 int main() {
9     Object o1;
10    Object o2 = o1;                                // copy
11    Object o3 = std::move(o1);                      // move
12    Object o4 = std::move_if_noexcept(o2);          // move
13
14    std::vector<Object> v(3);
15    std::println("-----");
16    v.resize(5);                                    // move, move, move 발생!
17    std::println("-----");
18 }
```

`std::move_if_noexcept`를 사용하면 컴파일러가 함수의 `noexcept` 키워드 유무를 검사하고 만약 키워드가 있다면 `move`를 실행한다. `std::move_if_noexcept`는 `type_traits` 기술로 예외 가능성을 조사한 후 예외 가능성이 있으면 `const T&` 타입으로 변환하고 예외 가능성이 없다면 `T&&` 타입으로 변환해주는 함수이다!

```
1 template<typename T>
2 constexpr std::conditional_t<
3     !std::is_nothrow_move_constructible_v<T> &&
4     std::is_copy_constructible_v<T>, const T&, T&&>
5     move_if_noexcept(T& x) noexcept {
6         return std::move(x)
7     }
```

다음과 같이 클래스가 여러 멤버 변수를 가지고 있는 경우를 살펴보자

```
1 template<typename T>
2 class Object{
3     int n;
4     std::string s;
5     T t;
6
7     public:
8     Object() = default;
9     Object(const Object& other) : n(other.n), s(other.s), t(other.t) {}
10    Object(Object&& other) noexcept
11        : n(other.n),
12        s(std::move(other.s)),
13        t(std::move(other.t))
14    {}
15};
```

복사 생성자를 보면 `int n`은 예외가 없다. 그리고 `std::string s`은 예외가 없음을 보장한다. 하지만 **임의의 타입 T는 예외가 존재할 가능성이 존재한다.** 이럴 땐 다음과 같이 키워드를 입력하면 된다.

```
1 Object(Object&& other) noexcept( noexcept( t(std::move(other.t)) ))
2     : n(other.n),
3     s(std::move(other.s)),
4     t(std::move(other.t))
5 {}
```

c++에서 `noexcept - 128) * 64 + (' - 128)` 꿀 두 가지 의미가 존재한다. 1) `noexcept operator`, 2) `noexcept specifier`

1. `noexcept operator`:

- `bool b = noexcept(expression)`: 어떤 표현식이 예외 가능성이 있는지 조사한다

2. noexcept specifier:

- `f() noexcept`, `f() noexcept(true)`: 함수 f는 예외가 없다.
- `f() noexcept(false)`: 함수 f는 예외가 존재한다

또 다른 방법으로는 다음과 같이 쓸 수 있다.

```
1 Object(Object&& other) noexcept( std::is_nothrow_move_constructible_v<T> )
2   : n(other.n),
3     s(std::move(other.s)),
4     t(std::move(other.t))
5   {}
```

2.13.4 Default move constructor

`move` 생성자를 기본적으로 컴파일러가 제공하는 경우에 대해 살펴보자

```
1 class Object{
2   std::string name;
3   public:
4     Object() = default;
5     Object(const Object& other) : name(obj.name) {}           // [1] 복사 생성자
6     Object& operator=(const Object& obj) { name = obj.name; } // [2] 복사 대입 연산자
7     Object(Object&& obj) : name(std::move(obj.name)) {}      // [3] move 생성자
8     Object& operator=(Object&& obj) { name = std::move(obj.name); } // [4] move 대입 연산자
9   };
```

1. **case 1.** 사용자가 [1,2,3,4] 모두 제공하지 않는 경우 컴파일러가 [1,2,3,4] 대한 default 버전을 제공한다.
2. **case 2.** 사용자가 [1] (또는 [2])만 제공하는 경우 컴파일러는 [2](또는 [1])는 default 버전을 제공하지만 [3,4]는 제공하지 않는다. 사실 [1]만 제공하면 컴파일러가 [2]도 제공하지 않는 것이 맞지만 설계 상 오류로 [2]는 default 버전이 제공된다. (since c++98)
3. **case 3.** 사용자가 [3](또는 [4])를 제공하는 경우 컴파일러는 [1], [2]를 삭제해버린다. 즉, 사용할 수 없다. 그리고 [4](또는 [3])는 제공하지 않는다.

하지만 코딩을 하다보면 복사 생성자는 사용자가 제공하지만 move 계열 함수만 컴파일러에게 요청하고 싶다.
이런 경우에는 어떻게 해야할까?

```
1 class Object{
2   std::string name;
3   public:
4     Object() = default;
5     Object(const Object& other) = default;      // 복사 대입 연산자도 move 생성자를 default로 요청하면
6     // 반드시 같이 요청해야 한다.
7     Object(Object&& obj) = default;
8     Object& operator=(Object&& obj) = default; // default 버전을 요청한다!
9   };
```

(...)=`default`;로 default 버전을 요청하면 컴파일러가 move 생성자는 `default` 버전을 생성한다. 복사 대입 연산자도 move 생성자를 `default`로 요청할 때 같이 요청해야 한다! 위와 같은 코드 형태는 잘 작성된 오픈소스에서 많이 볼 수 있다!

2.13.5 Rule of 3/5/0

다음으로 널리 통용되는 규칙인 Rule of 3/5/0에 대해 알아보자

```
1 class Person {
2   char* name;
3   int age;
4   public:
5     Person(const char*s, int a) :age(a) {
6       name = new char[strlen(s)+1];
7       strcpy(name, strlen(s)+1, s);
```

```

8   }
9   // char*와 같이 포인터 멤버변수가 있고 동적으로 메모리를 할당한다면
10  // c++98 시절에는 소멸자/복사 생성자/복사 대입연산자를 반드시 만들어야 했다 (Rule of 3)!
11  // c++11 시절에는 소멸자/복사 생성자/복사 대입연산자/ move 생성자/move 대입연산자 또한 만들어야 한다.
12  // (Rule of 5)
13 };

```

클래스 내부 변수에 포인터 멤버변수가 있고 동적으로 메모리를 할당한다면 반드시 선언해야 하는 함수를 가르켜 **Rule of 3 (c++98)**, **Rule of 5 (c++11)**라고 하였다.

`char*` 대신 `std::string`을 사용하면 사용자가 직접 자원을 관리할 필요가 없다. (= **Rule of 0**). 즉, STL에서 제공하는 클래스를 사용하면 클래스가 동적 메모리 할당에 대하여 컴파일러가 알아서 자동으로 제공한다. 결론은 STL을 많이 쓰면 좋다는 얘기이다.

2.14 Perfect forwarding

```

1 void foo(int n) {}
2 void goo(int& r) {r = 20; }
3
4 template<class F, class T>
5 void chronometry(F f, T arg) {
6     f(arg);
7 }
8
9 int main() {
10    int n = 10;
11    chronometry(foo, 10);
12    chronometry(goo, n); // error. T arg에 의해 값이 chronometry에 복사되어 goo에서 값을 20으로 바꿔도 n
13    // 값이 바뀌지 않는다.
14    std::cout << n << std::endl;
}

```

함수의 성능을 측정해주는 `chronometry` 템플릿 함수를 정의해보자. 이를 통해 두 함수 `foo`, `goo`를 하나의 함수 템플릿에서 처리하고 싶다. 이를 위해서는 perfect forwarding 기술이 필요하다. **Perfect forwarding**이란 전달 받은 인자를 다른 함수에게 “값, `const` 속성, value category 등의 변화없이 완벽하게 전달”하는 것을 말한다.

rvalue `n`과 lvalue `n`을 동시에 처리하기 위해 `const T&`를 사용할 수 있다. 하지만 이는 **전달 과정에서 const 속성이 추가되므로 완벽한 전달이라고 할 수 없다.**

```

1 void foo(const int n) {}
2 void goo(const int& r) { r=20; }
3
4 template<class F, class T>
5 void chronometry(F f, const T& arg) {
6     f(arg);
7 }

```

설명의 편의를 위해 우선 `T`를 `integer`로 고정해보자. lvalue와 rvalue를 속성 변화없이 그대로 받기 위해서는 다음과 같이 두 함수 템플릿이 필요하다.

```

1 template<class F, class T>
2 void chronometry(F f, int& arg) { // lvalue reference
3     f(arg);
4 }
5
6 template<class F, class T>
7 void chronometry(F f, int&& arg) { // rvalue reference
8     f(arg);
9 }

```

하지만 두 함수 템플릿은 `hoo()` 함수를 처리할 수 없다.

```

1 void hoo(int &&r) { }
2
3 template<class F, class T>
4 void chronometry(F f, int&& arg) { // rvalue reference
5     f(arg);
6 }

```

```

6   }
7
8   int main() {
9     hoo(10); // ok
10    chronometry(hoo, 10); // error
11 }
```

chronometry에서 rvalue reference `int&&`로 받아도 `arg`라는 이름이 생기기 때문에 더 이상 rvalue가 아니게 된다. 즉, 10은 rvalue이지만 `arg`는 lvalue이다.

```

1 template<class F, class T>
2 void chronometry(F f, int&& arg) { // rvalue reference
3   // f(arg);
4   f(static_cast<int&&>(arg)); // 다시 rvalue로 캐스팅해서 보내야 함!
5 }
```

따라서 rvalue를 호출할 때는 `static_cast`로 다시 한 번 rvalue로 캐스팅해서 넘겨줘야 한다.

2.14.1 Using forwarding reference

Forwarding reference를 사용하면 `int&`, `int&&`를 자동 생성할 수 있다. 따라서 함수 템플릿을 두 개 생성하지 않아도 된다. 하지만 템플릿을 사용하려면 구현부가 동일해야 하는데 하나는 캐스팅이 있고 하나는 캐스팅이 없다.

```

1 void chronometry(F f, int& arg) { // lvalue reference
2   f(static_cast<int&>(arg));
3 }
4
5 template<class F, class T>
6 void chronometry(F f, int&& arg) { // rvalue reference
7   f(static_cast<int&&>(arg));
8 }
```

따라서 lvalue에도 캐스팅을 추가해준다. 컴파일 타임에서 lvalue 캐스팅은 같은 타입 캐스팅이기 때문에 무시된다. 이제 구현부까지 동일해졌으므로 하나의 템플릿 함수로 합칠 수 있다.

```

1 template<class F, class T>
2 void chronometry(F f, T&& arg) { // rvalue reference
3   f(static_cast<T&&>(arg));
4 }
5
6 int main() {
7   chronometry(goo, n); // T = int&, T&& = int&
8   chronometry(hoo, 10); // T = int, T&& = int&&
9 }
```

Perfect forwarding은 `const`까지 자동으로 커버해주기 때문에 `const` 전용 함수를 만들지 않아도 된다. C++ 표준에는 캐스팅을 대신 해주는 함수가 존재한다. `static_cast`를 사용하는 대신 `std::forward` 함수를 사용한다.

```

1 template<class F, class T>
2 void chronometry(F f, T&& arg) { // rvalue reference
3   f( std::forward<T>(arg) );
4 }
```

`std::forward` 함수는 forward referencing을 자동으로 해주는 함수이며 lvalue를 (함수로 전달하면) lvalue로 캐스팅하고 rvalue를 (함수로 전달하면 받으면서 lvalue로 변경된 것을 다시) rvalue로 캐스팅해준다. 즉, 인자가 rvalue일 때만 `std::move()`를 하는 것을 의미한다!

2.14.2 Variadic parameter template

만약 `foo`와 `goo`의 파라미터 개수가 다르다면 어떻게 해야 할까? template에서 가변인자를 받는 `...`을 사용한다.

```

1 void foo() {}
2 int& goo(int a, int& b, int&& c) {
3   b=20;
```

```

4     return b;
5 }
6
7 template<class F, class ... T>
8 void chronometry(F f, T&& ... arg) { // rvalue reference
9     f( std::forward<T>(arg) ... );
10 }
```

goo는 리턴값이 존재한다. 이런 경우 어떻게 해야할까? `auto`를 사용하게 되면 참조값을 버리는 특성이 있으므로 `decltype(auto)`를 사용한다.

```

1 template<class F, class ... T>
2 decltype(auto) chronometry(F f, T&& ... arg) { // rvalue reference
3     return f( std::forward<T>(arg) ... );
4 }
```

Perfect forwarding을 정리하면 다음과 같다

1. 인자를 받을 때 **forwarding reference (`T&&`)**를 사용한다.
2. 인자를 다른 함수에 전달할 때 `std::forward<T>(arg)`로 묶어서 전달한다.
3. 여러 개의 인자를 모두 forwaring하기 위해 **가변인자 템플릿**을 사용한다.
4. 반환 값도 전달하기 위해서는 `decltype(auto)`로 반환한다.

2.14.3 Member function pointer for perfect forwarding

chronometry 함수에 멤버 함수를 전달할 수는 없을까?

```

1 void foo(int n) {}
2
3 class Test{
4     public:
5         void f1(int n) { cout << "Test t1" << endl; }
6     };
7
8 template<class F, class ... T>
9 decltype(auto) chronometry(F f, T&& ... arg){
10     return f(std::forward<T>(arg)...);
11 }
12
13 int main() {
14     chronometry(foo, 10); // ok
15
16     Test obj;
17     chronometry(&Test::f1, &obj, 10); // error. 이걸 가능하게 할 순 없을까?
18 }
```

`std::invoke`를 사용하면 일반함수 포인터는 `std::invoke(f, arg)`와 같이 호출하고 멤버함수 포인터는 `std::invoke(f, &obj, arg)`와 같이 호출한다.

```

1 void foo(int n) {}
2
3 class Test{
4     public:
5         void f1(int n) { cout << "Test t1" << endl; }
6     };
7
8 template<class F, class ... T>
9 decltype(auto) chronometry(F f, T&& ... arg){
10     return std::invoke(f, std::forward<T>(arg)...);
11 }
12
13 int main() {
14     chronometry(foo, 10); // ok
15 }
```

```

16     Test obj;
17     chronometry(&Test::f1, &obj, 10); // ok.
18 }
```

2.14.4 Function object for perfect forwarding

함수 객체 functor에 () 대입연산자를 재정의하면 함수처럼 쓸 수 있다.

```

1 struct Functor {
2     void operator()(int n) & { // lvalue 객체일 때 실행
3         cout << "operator()" &"<< endl;
4     }
5
6     void operator()(int n) && { // rvalue 객체일 때 실행
7         cout << "operator() &&"<< endl;
8     }
9 }
10
11 template<class F, class ... T>
12 decltype(auto) chronometry(F f, T&& ... arg){
13     return std::invoke(f, std::forward<T>(arg)...);
14 }
15
16 int main() {
17     Functor f;
18
19     chronometry(f, 10);           // ok. lvalue operator 정상적으로 호출
20     chronometry(Functor(), 10); // error. rvalue operator 가 호출되지 않고 lvalue opeartor가 호출됨
21     // (전달받을 때 복사본을 만들기 때문)
}
```

f를 넘겨줄 때 (F f)로 받고 있기 때문에 Functor()와 같이 rvalue를 호출해도 lvalue로 변환되는 것이다. 따라서 함수도 진짜 함수가 아닌 함수 객체가 올 수 있는데 이럴 경우 F 또한 forwarding reference(&&)로 넘겨줘야 한다. 그리고 받을 때도 std::forward<F>로 받아야 한다.

```

1 struct Functor {
2     void operator()(int n) & { // lvalue 객체일 때 실행
3         cout << "operator()" &"<< endl;
4     }
5
6     void operator()(int n) && { // rvalue 객체일 때 실행
7         cout << "operator() &&"<< endl;
8     }
9 }
10
11 template<class F, class ... T>
12 decltype(auto) chronometry(F&& f, T&& ... arg){
13     return std::invoke(std::forward<F> f, std::forward<T>(arg)...);
14 }
15
16 int main() {
17     Functor f;
18
19     chronometry(f, 10);           // ok
20     chronometry(Functor(), 10); // ok
21 }
```

2.14.5 Chronometry implementation

함수가 돌아가는데 걸리는 시간을 측정하기 위해 StopWatch 클래스를 작성해준다.

```

1 class StopWatch{ // in StopWatch.h
2 public:
3     StopWatch() : start(chrono::system_clock::now()) {}
```

```

4
5     ~StopWatch() {
6         end = chrono::system_clock::now();
7
8         chrono::duration<double> elapsed = end - start;
9
10        cout << elapsed.count() << " seconds..." << endl;
11    }
12
13    private:
14        chrono::system_clock::time_point start;
15        chrono::system_clock::time_point end;
16    };

```

chronometry 함수 내 한 줄만 추가해주면 시간을 측정하는 코드를 작성할 수 있다.

```

1 #include "StopWatch.h"
2
3 void foo(int n) {
4     std::this_thread::sleep_for(std::chrono::seconds(n));
5 }
6
7 template<class F, class ... T>
8 decltype(auto) chronometry(F&& f, T&& ... arg){
9     StopWatch sw; // 한 줄만 적으면 시간을 측정해준다.
10    return std::invoke(std::forward<F> f, std::forward<T>(arg)...);
11 }
12
13 int main() {
14     chronometry(foo, 2);
15 }

```

2.14.6 Notice for perfect forwarding

Perfect forwarding으로 인자를 넘겨줄 때 암시적인 중괄호 전달 대신 명시적으로 전달해야 한다.

```

1 void foo(std::pair<int, int> p) {}
2
3 int main() {
4     chronometry(foo, {1,2}); // error
5     chronometry(foo, std::pair{1,2}); // ok
6 }

```

일반 foo 함수를 호출할 때 foo(1,2)는 정상적으로 동작하지만 perfect forwarding에서는 명시적으로 타입을 정의해줘야 한다.

같은 이름의 함수가 오버로딩된 경우 perfect forwarding 시 함수의 모양을 캐스팅해줘야 한다.

```

1 void goo(int a) {}
2 void goo(int a, int b) {}
3
4 int main(){
5     chronometry(goo, 1, 2); // error. goo를 넘겨주는 시점에서 어떤 함수를 넘겨줘야 할지 모른다.
6
7     chronometry(static_cast<void(*)(int, int)>(goo), 1,2); // ok
8 }

```

2.14.7 Perfect forwarding in STL

이번 섹션에서는 perfect forwarding 기술이 stl에서 어떻게 사용되고 있는지 살펴본다.

```

1 class Point{
2     int x,y;
3     public:
4     Point(int x, int y) :x(x), y(y) {}
5     Point(const Point& pt) :x(pt.x), y(pt.y) {}

```

```

6     ~Point() {}
7 };
8
9 int main() {
10    vector<Point> v;
11
12    Point pt(1,2);
13    v.push_back(pt); // 이 순간 어떤 일이 발생할 것인가?
14
15    cout << "-----" << endl;
16 }

```

v.push_back(pt)가 호출되는 순간 v 내부 버퍼에 사용자가 전달한 Point 객체의 복사본을 생성한다. 즉, 12번 줄에서 Point 객체를 한 번 생성하고 13번 줄에서 push_back이 실행되면서 Point 객체가 복사로 인해 한 번 더 생성된다.

```

1 int main() {
2    vector<Point> v;
3
4    Point pt(1,2); // Point(int, int) 호출
5    v.push_back(pt); // Point(const Point&) 호출
6
7    cout << "-----" << endl;
8    // 소멸자 2개 호출
9 }

```

다른 방법으로 임시객체를 만들어보자.

```

1 int main() {
2    vector<Point> v;
3
4    v.push_back(Point(1,2)); // 임시 객체 호출하면 소멸자가 불리는 시점만 달라질 뿐 2개가 생성되는 것은
5    // 동일
6    // Point 임시 객체는 바로 파괴됨
7
8    cout << "-----" << endl;
9    // v에 들어간 Point 객체의 소멸자 호출
}

```

좀 더 효율적인 방법은 없을까? `emplace_back`을 사용하면 객체 자체를 전달하지 않고 객체를 생성할 때 필요한 데이터만 전달할 수 있다.

```

1 int main() {
2    vector<Point> v;
3
4    v.emplace_back(1,2); // 생성자 1개만 호출
5
6    cout << "-----" << endl;
7    // v에 들어간 Point 객체의 소멸자 호출
8 }

```

```

1 template<class ... Args>
2 constexpr reference emplace_back(Args&& ... args); // since c++20 (from cppreference.com)

```

Type이 Point로 변환되고 args에 (1,2)가 전달되어 호출된다. 만약 생성자가 lvalue reference 또는 rvalue reference를 입력으로 받아도 `emplace_back`은 정상적으로 동작해야 한다. 이를 위해서 `emplace_back`는 **perfect forwarding 기술**을 사용하여 설계되었다.

```

1 template<typename ... Ts>
2 decltype(auto) emplace_back(Ts&& ... args) {
3     Type* t = new Type(std::forward<Ts>(args) ...);
4     return *t;
5 }

```

따라서 stl 컨테이너에서 사용자 정의 타입을 값(value)로 보관할 때는 push 계열의 함수 대신 emplace 계열의

함수를 사용하는 것이 좋다.

2.15 Callable object

2.15.1 Function object

```
1 struct plus{
2     int operator()(int arg1, int arg2) { return arg1 + arg2; }
3 };
4
5 int main() {
6     plus p;
7     int n = p(1,2);      // p는 객체이지만 함수처럼 사용하고 있다.
8     cout << n << endl;
9 }
```

함수 객체란 괄호 ()를 사용해서 호출 가능한 객체를 말한다. a와 b가 객체일 때 두 객체 사이의 연산은 컴파일러가 아래와 같이 변환해준다. 이를 **연산자 재정의**라고 한다.

```
1 a + b    // a.operator+(b)
2 a - b    // a.operator-(b)
3 a()       // a.operator()()
4 a(1,2)   // a.operator()(1,2)
```

함수 대신 함수 객체를 사용하는 이유가 뭘까?

1. 함수와 달리 상태를 가질 수 있다.
2. 특정 상황에서 함수보다 빠르게 사용할 수 있다(인라인 치환).
3. 모든 함수 객체는 자신만의 타입을 가진다.

앞서 정의한 plus 객체를 완성도 있게 만들어보자.

```
1 template<class T>
2 struct plus {
3     [[nodiscard]] constexpr
4     T operator()(const T& arg1, const T& arg2) const {
5         return arg1 + arg2;
6     }
7 };
8
9 int main() {
10     const plus<int> p; // template으로 다양한 타입들을 사용할 수 있고 const plus 객체 또한 const 구문을
11     // 추가함으로써 사용할 수 있다.
12     p(3,4);           // [[nodiscard]]에 의해 컴파일 경고문이 뜬다.
13     int n = p(1,2);
14     cout << n << endl;
15 }
```

함수 객체를 작성할 때 권장되는 가이드는 다음과 같다.

1. 템플릿으로 만들어서 다양한 타입들이 호환 가능하도록 할 것
2. `operator()`는 `const` 멤버 함수로 작성할 것
3. `constexpr`, `[[nodiscard]]`를 사용하여 컨벤션을 강화할 것
4. `noexcept`이 필요한 경우 사용할 것
5. perfect forwarding & template specialization 기술을 적용할 것
6. `is_transparent` 멤버 함수가 필요하면 추가할 것

2.15.2 Function object = function with state

함수가 상태를 가진다는 말은 무엇을 의미할까? 다음 예제를 보자.

```
1 int urand() {
2     return rand() % 10;
3 }
4
5 int main() {
6     for(int i=0; i<10; i++){
7         cout << urand() << ", ";
8     }
9     cout << endl;
10 }
```

위 함수를 실행하면 난수가 생성된다. 이 때 중복되는 난수가 생성될 수 있는데 만약 중복되지 않은 난수를 구하려면 어떻게 해야 할까? 그러기 위해서는 한 번 반복했던 난수 값은 어딘가에 보관되어야 한다. 하지만 함수는 '동작은 있으나 상태는 없으므로' 한 번 반복한 값은 잊어버리고 다시 새롭게 동작한다. 반면에 **함수 객체를 사용하면 함수의 난수 값을 상태 변수에 기억하여 중복되지 않게 호출할 수 있다.**

```
1 class URandom {
2     // member data에 저장하여 상태를 기억할 수 있다!
3     public:
4         int operator()() {
5             return rand() % 10;
6         }
7     };
8
9     int main () {
10         URandom urand;
11         for(int i=0; i<10; i++){
12             cout << urand() << ", ";
13         }
14         cout << endl;
15     }
```

클래스를 완성도 있게 작성해보자

```
1 class URandom{
2     std::bitset<10> bs; // 10-bit을 관리하기 위한 멤버 변수
3     bool recycle;
4
5     std::mt19937 rancgen { std::random_device{}() };
6     std::uniform_int_distribution<int> dist{0, 9};
7     public:
8     URandom(bool recycle = false) : recycle(recycle) {
9         // bs.set(5); // 5번째 비트만 1로
10        bs.set(); // 모든 비트를 1로
11    }
12
13    int operator()(){
14        if(bs.none()) { // 모든 값을 꺼낸 경우 recycle 변수에 따라 다시 순행하거나 -1 리턴
15            if(recycle) bs.set();
16            else return -1;
17        }
18
19        int k=-1;
20        while( !bs.test( k = dist(rancgen) )); // 난수를 구하고 그 값이 bs에 1로 세팅되어 있는지 확인. 0
21        // 이면 이미 나온 값이므로 다른 값을 난수로 구함
22        bs.reset(k); // 처음 나온 값은 0으로 세팅
23
24        return k;
25    }
26};
```

2.15.3 closure & function object

```
1  bool f1(int a) { return a%3 == 0; }
2
3  int main() {
4      std::vector<int> v = { 1,2,6,7,8,3,4,5,9,10};
5      auto r1 = std::find(v.begin(), v.end(), 3);
6      auto r2 = std::find_if(v.begin(), v.end(), f1);
7  }
```

std::find는 주어진 컨테이너에서 값을 바로 찾는 함수이고 std::find_if는 주어진 컨테이너에서 조건자(predicate)에 맞는 값을 찾는 함수이다. 조건자(predicate)란 bool(또는 bool로 변환 가능한 타입)을 반환하는 함수 또는 함수 객체를 말한다.

위 코드에서 f1은 3의 배수를 찾는 함수였다. 만약 사용자로부터 입력받는 정수 k의 배수를 검색하게 하고 싶다면 어떻게 해야 할까? 전역 변수를 쓰면 해결할 수 있지만 이런 코드를 작성할 때마다 전역 변수를 생성한다면 매우 비효율적인 코딩이 된다. 함수 객체를 사용하면 이를 쉽게 해결할 수 있다.

```
1  class F{
2      int value;
3  public:
4      F(int v) : value(v) {}
5
6      bool operator()(int n) const {
7          return n % value == 0;
8      }
9  };
10
11 int main() {
12     std::vector<int> v = { 1,2,6,7,8,3,4,5,9,10};
13     int k = 3;
14     auto r1 = std::find_if(v.begin(), v.end(), F(k)); // 함수 객체를 사용하여 사용자로부터 입력받은 k의
15     배수를 쉽게 구할 수 있다.
16     cout << *r1 << endl;
}
```

위와 같이 함수 객체는 scope 내에 지역 변수를 캡처할 수 있는 기능이 있는데 이를 closure라고 한다. 프로그래밍 분야에서 closure의 자세한 정의는 scope 내의 지역 변수를 바인딩 할 수 있는 일급함수객체(first-class object)이고 1960년대 처음 용어가 등장하였다.

2.15.4 inline & function object

인라인.inline) 함수는 컴파일 시에 함수 호출식을 함수의 기계어 코드로 치환하는 것을 의미한다.

```
1  int add1(int a, int b) { return a + b; }
2  inline int add2(int a, int b) { return a + b; }
3
4  int main () {
5      int ret1 = add1(1,2); // 함수 호출
6      int ret2 = add2(1,2); // 치환
7
8      int(*f)(int, int) = &add2; // add2를 함수 포인터로 사용하면 호출될까?
9      f(1, 2);
10 }
```

인라인 함수는 컴파일 타임에 치환되기 때문에 함수 포인터 (*f)를 통해 인라인 함수 add2를 가리키면 치환되지 않고 호출된다. (*f)는 런타임 시 수시로 가리키는 포인터가 달라질 수 있으므로 예측하기 어렵기 때문이다.

다른 예제를 보자. sort 함수의 비교 정책을 함수 포인터로 바꾸고 싶은데 속도 향상을 위해 cmp를 인라인 함수로 치환하면 정말 속도가 빨라질까?

```
1  void sort(int* x, int sz, bool(*cmp)(int, int)) { // 함수 포인터를 사용하여 비교 정책 변경
2      for(int i=0; i<sz-1; i++){
3          for(int j=i+1; j<sz; j++){
4              if(x[i] > x[j])
5                  if(cmp(x[i], x[j]))
6                      std::swap(x[i], x[j]);
7      }
8  }
```

```

7         }
8     }
9 }
10
11    inline bool cmp1(int a, int b) {return a<b;}
12    inline bool cmp2(int a, int b) {return a>b;}
13
14    int main() {
15        int x[10] = {1,3,5,7,9,2,4,6,8,10};
16        sort(x, 10, &cmp1);
17        sort(x, 10, &cmp2); // 인라인 치환이 되었을 거라고 예상했으나 함수 포인터로 인라인 함수를 호출하면
18        치환되지 않는다!
}

```

cmp1과 cmp2는 다른 함수지만 함수 포인터로 가리키게 되면 signature(반환 타입과 인자 모양)이 같으므로 동일한 함수 타입으로 간주된다. 즉, 함수는 자신만의 타입이 없다. 반면에 함수 객체는 signature가 동일해도 클래스 이름으로 구분할 수 있다. 따라서 함수 객체는 자신만의 타입이 있다.

```

1    inline bool cmp1(int a, int b) {return a<b;}
2    inline bool cmp2(int a, int b) {return a>b;}
3
4    struct Less {
5        inline bool operator()(int a, int b) const { return a < b; }
6    };
7
8    struct Greater{
9        inline bool operator()(int a, int b) const { return a > b; }
10   };
11
12   template<class T>
13   void sort(int* x, int sz, T cmp) {
14       for(int i=0; i<sz-1; i++){
15           for(int j=i+1; j<sz; j++){
16               if(cmp(x[i], x[j]))
17                   std::swap(x[i], x[j]);
18           }
19       }
20
21       int main(){
22           int x[10] = {1,3,5,7,9,2,4,6,8,10};
23           Less f1;
24           Greater f2;
25
26           sort(x, 10, &cmp1);
27           sort(x, 10, &cmp2);
28           sort(x, 10, f1);
29           sort(x, 10, f2);
30       }
31   }

```

sort 비교 정책으로 일반 함수(cmp1, cmp2)를 사용하면 signature가 동일하기 때문에 비교 정책을 교체해도 코드 메모리가 증가하지 않는다. 하지만 비교 정책 함수가 인라인 치환될 수 없기 때문에 느린다. 이와 반대로 sort 비교 정책으로 함수 객체(Less, Greater)를 사용하면 비교 정책이 인라인 치환되기 때문에 빠르지만 정책을 교체한 횟수 만큼의 sort() 함수가 생성되어 코드 메모리가 증가한다.

2.15.5 STL & function object

c++ 표준에는 이미 많은 함수 객체를 제공한다. 이는 functional 헤더에 포함되어 있다.

```

1 #include <functional>
2
3 int main() {
4     int x[10] = {1,3,5,7,9,2,4,6,8,10};
5
6     std::greater<int> f;
7     std::sort(x, x+10, f);

```

```

8     std::sort(x, x+10, std::greater<int>()); // 임시 객체 표기법
9     std::sort(x, x+10, std::greater());      // c++17부터 타입 생략 가능
10    std::sort(x, x+10, std::greater{});       // 중괄호 표기법 가능
11

```

3 STL programming

3.1 STL structure

3.1.1 Generic algorithm - find

문자열 내에서 원하는 문자를 찾는 strchr 함수를 작성해보자

```

1  char* mystrchr(char* s, int c) {
2      while(*s != c) {
3          if(!*s++) return nullptr;
4      }
5      return s;
6  }
7
8  int main() {
9      char s[] = "abcdefg";
10
11     char *p = mystrchr(s, 'c');
12
13     if(p == nullptr) std::println("fail");
14     else             std::println("success : {}", *p);
15 }

```

전체 문자열이 아니라 부분 문자열 검색을 가능하게 할 수는 없을까? 문자열 "시작 뿐 아니라 끝도 전달"해야 한다

- 방법1: 시작 주소와 요소의 개수 전달
- 방법2: 시작 주소와 마지막 주소를 전달

이 중 c++에서는 방법2를 채택하여 사용하고 있다. 구간의 끝(last)을 검색 대상에 포함할 것인가? 포함하지 않는 것이 장점이 많다

- s = [a,b,c,d,e,f,g]과 같이 주어졌을 때 first=s, last=s+4인 경우 [a,b,c,d]만이 문자열 구간에 포함된다. 이 때, s='a', s+4='e'이다.
- 이를 [first, last)와 같이 표기하며 반개행 구간(half open range)라고 부른다.

```

1  char* mystrchr(char* first, char* last, int c) {
2      for(; first != last; ++first) {
3          if(*first == c) return first;
4      }
5      return nullptr;
6  }
7
8  int main() {
9      char s[] = "abcdefg";
10
11     char *p = mystrchr(s, s+4, 'e');
12
13     if(p == nullptr) std::println("fail");
14     else             std::println("success : {}", *p);
15 }

```

- 위 코드에서는 s='a', s+4='e'를 가르키고 있으므로 e는 문자열 구간에 포함되지 않는다. 따라서 fail이 출력된다.

위 코드를 일반화하여 문자열 뿐만 아니라 모든 타입의 배열에서 선형 검색을 하는 함수를 작성해보자. 템플릿 (template) 프로그래밍을 하자는 의미이다.

```

1 template<typename T>
2 T* find(T* first, T* last, T c) {      // 모든 배열에서 원소를 찾으므로 이름을 일반화하여 find로 명명한다
3     for(; first != last; ++first) {
4         if(*first == c) return first;
5     }
6     return nullptr;
7 }

8
9 int main() {
10    char s[] = "abcdefg";
11    char* p = find(s, s+4, 'e');          // char 뿐만 아니라
12
13    double d[] = {1,2,3,4,5,6,7,8,9,10};
14    double* pd = find(d, d+10, 5.0);     // double 타입도 작동!
15
16    if(p == nullptr) std::println("fail");
17    else             std::println("success : {}", *p);
18}

```

- 혹자는 10개 크기의 배열을 만들고 11번째 주소를 레퍼런싱하는게 말이 되냐고 물을 수 있는데 c++ 표준문서에 보면 비교 연산을 위해 n개 크기의 배열의 n+1 주소를 사용하는 것은 문제없다고 되어 있다!

위 코드에는 다소 황당한 컴파일 에러가 존재한다.

```

1 template<typename T>
2 T* find(T* first, T* last, T c) {
3     for(; first != last; ++first) {
4         if(*first == c) return first;
5     }
6     return nullptr;
7 }

8
9 int main() {
10    double d[] = {1,2,3,4,5,6,7,8,9,10};
11    double* pd = find(d, d+10, 5);        // error! double형 타입이 아닌 int형 타입은 못찾음!
12    double* pd = find(d, d+10, 5.0f);    // error! double형 타입이 아닌 float형 타입은 못찾음!
13}

```

- double 배열에서 int, float을 못찾는 것은 말이 안된다([치명적 단점](#)).

- 함수 인자로 T*로 표기하면 Raw Pointer 사용 가능하다. 이는 즉, 포인터처럼 작동하는 객체들(스마트 포인터, 반복자 등)을 사용할 수 없다는 의미.

- 해결책1: 구간을 나타내는 타입(T1)과 검색 대상 타입(T2)를 분리한다.
- 해결책2: T* 대신 T1으로 표기해서 구간을 나타내는 타입이 "[반드시 포인터이어야 한다는 조건을 제거](#)"한다. 단, 구간을 나타내는 타입은 ==, !=, ++, * 연산이 가능해야 한다.

```

1 template<typename T1, typename T2>
2 T1 find(T1 first, T1 last, const T2& value);

```

- T1 first, T1 last → 구간을 나타내는 타입으로 포인터 또는 반복자가 올 수 있다. 일반적으로 "call by value"로 받는다

- const T2 value → 검색할 요소. user type이 될 수도 있다. 복사본의 오버헤드를 줄이기 위해 일반적으로 "const reference"로 받는다

현재 코드는 검색 실패 시 nullptr를 반환한다. nullptr 대신 "last"를 반환하게 하면 어떨까? "last"는 검색 대상이 아니므로 검색 성공 시 last가 반환될 수는 없다. [first, last) 구간의 검색 실패 시 "[last는 다음 구간의 시작](#)"으로 사용될 수 있다.

```

1 template<typename T1, typename T2>
2 constexpr T1 find(T1 first, T1 last, const T2& value);
3 for(; first != last; ++first) {
4     if(*first == value) return first;

```

```

5     }
6     return last; // 검색 실패 시 nullptr이 아닌 last를 반환!
7 }
8
9 int main() {
10    double d[] = {1,2,3,4,5,6,7,8,9,10};
11    double* p = find(d, d+5, 7);
12
13   if(p == d+5) std::println("fail");
14   else         std::println("success : {}", *p);
15 }
```

- 함수 앞에 constexpr을 붙이면 컴파일 시간에 수행되어 성능이 좋아진다.

지금까지 작성한 find 함수는 c++ STL에 <algorithm> 헤더에 미리 구현되어 있다(c++98). 지금까지 구현한 find 함수처럼 템플릿 기반의 함수들을 "Generic Algorithm"이라고 부른다.

- 생각해 볼 문제 → 주어진 구간에서 "특정 값"이 아닌 "조건을 검색"하게 할 수 없을까? ex) [first, last) 구간에서 첫번째 나오는 짝수를 찾고 싶다.

3.1.2 Make iterator

다음과 같은 Single Linked List 코드를 보자.

```

1 template<typename T> struct Node {
2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
7 template<typename T> class slist {
8     Node<T>* head=nullptr;
9     public:
10    void push_front(const T& data) {
11        head = new Node<T>(data, head); // 아래와 같이 입력된다.
12        // 10, 0
13        // 20, 100
14        // 30, 200
15        // 40, 300
16        // 50, 400
17    }
18 };
19
20 int main() {
21     slist<int> s;
22     s.push_front(10);
23     s.push_front(20);
24     s.push_front(30);
25     s.push_front(40);
26     s.push_front(50);
27 }
```

- 주제에 집중하기 위해 "move 지원, 요소 제거" 등은 생략하고 push_front() 멤버 함수만 제공한다.

slist도 배열처럼 여러 개의 값을 보관하고 있다. slist에서 값을 검색하기 위해 앞서 만든 find() 알고리즘을 사용할 수 있을까? → 배열이 아니므로 불가능!

- 배열 - [50,40,30,20,10]
- 리스트 - [50, 0x400] → [40, 0x300] → [30, 0x200] → [20, 0x100] → [10, 0x0]

p,q가 각각 배열과 리스트의 첫번째 요소를 가리키는 포인터일 때

다음 요소로 이동	<code>++p</code>	<code>q = q->next</code>
요소에 접근	<code>*p</code>	<code>q->data</code>

위와 같이 두 자료구조는 이동, 접근 방식이 전부 다르다! 앞서 작성한 find() 알고리즘은 포인터에 맞춰 작성되어 있기 때문에 slist 리스트에는 사용할 수 없다.

일관성을 위해 리스트에서도 ++q, *q 연산자가 동작해야 한다. 이를 위해 반복자(iterator)를 별도로 작성해야 한다.

```
1 template<typename T> struct Node {
2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
7 template<typename T> class slist_iterator {           // 반복자(iterator)
8     Node<T>* current;
9 public:
10    slist_iterator(Node<T>* p = nullptr) : current{p} {}
11
12    slist_iterator& operator++() {                   // ++p, *p, p==q, p!=q를 위한 연산자 재정의
13        current = current->next;
14        return *this;
15    }
16    T& operator*() {
17        return current->data;
18    }
19    bool operator==(const slist_iterator& it) const { return current == it.current; }
20    bool operator!=(const slist_iterator& it) const { return current != it.current; }
21 };
22
23 slist_iterator<int> p{0x500};
24 ++p;          // ok. 연산자 동작
25 int n = *p;  // ok. 연산자 동작
```

3.1.3 Generic Algorithm - list

이전 섹션에서 만든 코드를 find() 알고리즘과 섞어보자. find를 사용하여 slist의 모든 요소를 검색하려면 첫번째 요소를 가리키는 반복자와 "마지막 다음 요소"를 가리키는 반복자가 필요하다.

```
1 template<typename T> struct Node {
2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
7 template<typename T> class slist_iterator {           // 반복자(iterator)
8     Node<T>* current;
9 public:
10    slist_iterator(Node<T>* p = nullptr) : current{p} {}
11
12    slist_iterator& operator++() {                   // ++p, *p, p==q, p!=q를 위한 연산자 재정의
13        current = current->next;
14        return *this;
15    }
16    T& operator*() {
17        return current->data;
18    }
19    bool operator==(const slist_iterator& it) const { return current == it.current; }
20    bool operator!=(const slist_iterator& it) const { return current != it.current; }
21 };
22
23 template<typename T> class slist {
24     Node<T>* head=nullptr;
25 public:
26    void push_front(const T& data) {
27        head = new Node<T>(data, head);
28    }
29 }
```

```

30     slist_iterator<T> begin() { return slist_iterator<T>{head}; } // begin(), end() 함수 제공해야함!
31     slist_iterator<T> end() { return slist_iterator<T>{nullptr}; }
32 };
33
34 int main() {
35     slist<int> s;
36     s.push_front(10);
37     s.push_front(20);
38     s.push_front(30);
39     s.push_front(40);
40     s.push_front(50);
41
42     slist_iterator<int> p1 = s.begin(); // 반복자 받은 후
43     slist_iterator<int> p2 = s.end();
44
45     while(p1 != p2) {           // 모든 요소 순회
46         std::print("{} , ", *p1);
47         ++p1;
48     }
49 }
```

- STL 규칙 - 모든 컨테이너는 반드시 `begin()`, `end()` 멤버 함수가 제공되어야 한다.

컨테이너와 반복자의 클래스 이름

- 컨테이너 클래스의 이름은 C++ 표준에 의해서 정해져 있다. ex) `vector`, `queue`, `list`, ...
- 하지만 반복자의 클래스 이름은 정해져 있지 않다.

따라서 컨테이너 설계자가 지켜야 하는 규칙이 있다. 반드시 자신의 반복자 클래스 이름은 "iterator"라는 약속된 이름으로 외부에 노출하여야 한다.

```

1 template<typename T> class slist {
2     Node<T>* head=nullptr;
3
4     public:
5         using iterator = slist_iterator<T>;           // 반복자는 iterator라는 이름으로 사용할 수 있다.
6
7         void push_front(const T& data) {
8             head = new Node<T>(data, head);
9         }
10
11         iterator begin() { return iterator{head}; } // 임의의 이름 -> iterator로 변경!
12         iterator end() { return iterator{nullptr}; }
13     };
14
15     int main() {
16         slist<int> s;
17         s.push_front(10);
18         s.push_front(20);
19         s.push_front(30);
20         s.push_front(40);
21         s.push_front(50);
22
23         slist<int>::iterator p1 = s.begin(); // 반복자는 다음과 같이 호출한다! 일반적으로 auto를 사용하여 이를
24         생략할 수 있다.
25         slist<int>::iterator p2 = s.end();
26
27         while(p1 != p2) {
28             std::print("{} , ", *p1);
29             ++p1;
30         }
31     }
32 }
```

반복자까지 구현되어 있으면 `find()` 알고리즘이 정상적으로 동작한다.

```

1 template<typename T> struct Node {
2     T data;
```

```

3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
7 template<typename T> class slist_iterator {
8     Node<T>* current;
9     public:
10    slist_iterator(Node<T>* p = nullptr) : current{p} {}
11
12    slist_iterator& operator++() {
13        current = current->next;
14        return *this;
15    }
16    T& operator*() {
17        return current->data;
18    }
19    bool operator==(const slist_iterator& it) const { return current == it.current; }
20    bool operator!=(const slist_iterator& it) const { return current != it.current; }
21 };
22
23 template<typename T> class slist {
24     Node<T>* head=nullptr;
25     public:
26     using iterator = slist_iterator<T>;
27
28     void push_front(const T& data) {
29         head = new Node<T>(data, head);
30     }
31
32     iterator begin() { return iterator{head}; }
33     iterator end() { return iterator{nullptr}; }
34 };
35
36 template<typename T1, typename T2>
37 constexpr T1 find(T1 first, T1 last, const T2& value);
38 for(; first != last; ++first) {
39     if(*first == value) return first;
40 }
41 return last;
42 }
43
44 int main() {
45     slist<int> s;
46     s.push_front(10);
47     s.push_front(20);
48     s.push_front(30);
49     s.push_front(40);
50     s.push_front(50);
51
52     auto ret = find(s.begin(), s.end(), 30); // 반복자 덕분에 find() 알고리즘이 정상적으로 동작한다
53
54     if(ret == s.end()) std::println("fail");
55     else             std::println("{}", *ret);
56 }

```

- 배열 뿐만 아니라 모든 컨테이너에서 선형 검색을 수행할 수 있다. ([진정한 Generic Algorithm이 되었다](#))

3.1.4 Container, Iterator, Algorithm (CIA)

컨테이너 안에 1이 몇 개 있는지 알고 싶다.

```

1 int main() {
2     std::vector c{1,2,3,1,2,3,1,2,3,1};
3     std::list   c{1,2,3,1,2,3,1,2,3,1};
4

```

```

5 int n = std::count(c.begin(), c.end(), 1); // std::count 알고리즘을 사용하면 된다. 반복자만 전달해주면
6   vector, list 상관없이 사용 가능하다.
7
8 std::println("{}",n);
}

```

알고리즘의 인자로 "반복자"가 아닌 "컨테이너"를 보낼 수는 없을까?

```

1 int main() {
2   std::vector c{1,2,3,1,2,3,1,2,3,1};
3
4   int n1 = std::count(c.begin(), c.end(), 1); // c++98 스타일
5
6   int n2 = std::ranges::count(c, 1);           // c++20 스타일
7   int n3 = std::ranges::count(c.begin(), c.end(), 1);
8
9   using namespace std::ranges;                 // namespace 생략
10  int n4 = count(c, 1);
11  int n5 = count(c.begin(), c.end(), 1);
12
13 std::println("{}",n);
14 }

```

- c++98 시절에는 기술적인 문제로 컨테이너를 인자로 받는 알고리즘을 만들 수 없었다.
- c++20에 concept이 추가되면서 std::ranges에 컨테이너를 인자로 받는 알고리즘들이 대부분 구현되어 있다.

3.1.5 Member type

T가 컨테이너일 때 요소의 타입을 알고 싶다.

```

1 template<typename T, typename Ax = std::allocator<T>>
2 class list {
3 public:
4   using value_type = T;    // 컨테이너 규칙 상 value_type으로 타입을 외부에 알려줘야 한다!
5 ...
6 }
7
8 list<int> s = {1,2,3,4,5};
9 list<int>::value_type n = s.front();

```

다음 예제와 같이 value_type은 함수 안에서 유용하게 사용할 수 있다.

```

1 template<typename T>
2 void print_first_element(const T& v) {
3   typename T::value_type n = v.front(); // typename을 앞에 추가해줘서 값이 아니라 타입임을 명시한다!
4   std::println("{}", n);
5 }
6
7 int main() {
8   std::list<int> c{1,2,3,4,5};
9   print_first_element(c);
10 }

```

T::value_type 앞에 typename은 왜 붙이는 걸까? 다음 코드를 보자.

```

1 struct Test {
2   static constexpr int value = 10;
3   using DWORD = int;
4 };
5 int p1 = 0;
6
7 template<typename T>
8 void foo(T a) {
9   // 아래 코드에서 * 연산자의 의미는?
10  T::value* p1; // 곱셈

```

```
11 T:::DWORD* p2; // 포인터 변수 선언  
12 }
```

- T:: 다음에는 값 또는 타입이 올 수 있다. 값이 오면 *는 곱셈이 되고 타입이 오면 *는 포인터 변수의 선언이 된다!

dependent name

- 템플릿 인자 T에 의존해서 꺼내는 이름 (T::Name)
- T는 임의의 타입이므로 컴파일러가 Name의 의미를 조사할 수 없다
- 따라서 타입은 앞에 `typename T::` 을 사용하여 타입이라고 명시해줘야 한다!

```
1 struct Test {  
2     static constexpr int value = 10;  
3     using DWORD = int;  
4 };  
5     int p1 = 0;  
6  
7     template<typename T>  
8     void foo(T a) {  
9         T::value* p1;  
10        typename T::DWORD* p2; // typename 선언  
11 }
```

3.2 Iterator

3.2.1 Iterator concept

반복자(iterator)란 컨테이너의 "내부구조에 상관없이 동일한 방법으로 모든 요소에 순차적으로 접근"하기 위한 객체이다. 반복자를 사용하면 컨테이너의 내부구조를 몰라도 컨테이너의 모든 요소에 동일한 방법으로 접근할 수 있다.

STL의 반복자는 포인터와 동일한 방법으로 사용할 수 있도록 설계되어 있다. 포인터와 동일하게 `++p`, `p++`, `*p`를 사용하여 증가, 감소, 요소에 접근할 수 있다.

```
1 int main() {  
2     std::list s{1,2,3,4};  
3     std::vector v{1,2,3,4}; // list, vector는 서로 다른 자료구조이지만  
4  
5     auto si = s.begin(); // 동일한 방법으로 반복자를 얻어서  
6     auto vi = v.begin();  
7  
8     ++si, ++vi; // 동일한 방법으로 이동하고 요소에 접근할 수 있다.  
9     std::println("{} {}", *si, *vi);  
10 }
```

- 객체가 iterator 요구 조건을 만족하는지 확인하는 방법
- c++20에서 추가된 concept 라이브러리를 사용하면 된다.
- `#include <iterator>`

Concept	요구 조건
indirectly_readable	<code>=*it</code>
indirectly_writable	<code>*it =</code>
weakly_incrementable	<code>++it && it++</code>
incrementable	<code>weakly_incrementable && regular</code>
input_iterator	<code>weakly_incrementable && indirectly_readable</code>
output_iterator	<code>weakly_incrementable && indirectly_writable</code>
input_or_output_iterator	<code>input_iterator output_iterator</code>

다음은 concept을 사용하여 반복자의 타입을 판별하는 코드이다.

```

1 template<typename T>
2 void check(const T& it) {
3     std::println("{}", std::input_or_output_iterator<T> );
4 }
5
6 int main() {
7     int x[4]{1,2,3,4};
8     std::vector v{1,2,3,4};
9
10    int* p1 = x;
11    auto p2 = v.begin();
12
13    check(p1); // true
14    check(p2); // true
15
16    int n=0;
17    check(n); // false
18 }
```

3.2.2 Container iterator

컨테이너의 반복자 타입은 컨테이너 이름<타입>::iterator와 같이 선언한다.

```

1 std::vector<int>::iterator p = c.begin(); // 코드 변경 시 해당 줄도 변경해야 하므로 auto 사용이 권장된다
2 auto p = c.begin();
```

컨테이너 반복자의 end()는 마지막이 아닌 "마지막 다음 요소"를 가리킨다. (Past-the-last element)

- * 연산자로 요소에 접근하면 안된다.
- 반복문 등에서 끝에 도달했는지 확인하는 용도로 사용된다.

컨테이너에서 반복자를 꺼내는 3가지 방법은 다음과 같다

```

1 auto p1 = c.begin();           // c++98, c가 배열이라면 사용 불가능
2 auto p2 = std::begin(c);      // c++11, c가 배열이라도 사용 가능
3 auto p3 = std::ranges::begin(c); // c++20, 기존 방법보다 안전한 방법
```

```

1 void f1(auto& c) { // c가 배열이라면 사용 불가능
2     auto p = c.begin();
3     auto s = c.size();
4 }
5
6 void f2(auto& c) {
7     auto p = std::begin(c);
8     auto s = std::size(c);
9 }
10
11 int main() {
12     std::vector c{1,2,3,4};
13     int c[] {1,2,3,4};
14
15     f1(c); // c가 배열이라면 사용 불가능!
16     f2(c); // 모든 c에 대해 사용 가능
17 }
```

STL의 std::begin()는 컨테이너와 배열 버전이 각각 구현되어 있다.

```

1 template<typename C>
2 constexpr auto mybegin(C& c) noexcept(noexcept(c.begin()) -> decltype(c.begin())) { // c가 컨테이너인 경우
3     return c.begin();
4 }
5
6 template<typename T, std::size_t SZ> // c가 배열인 경우
```

```

7 constexpr T* mybegin(T(&arr) [SZ]) noexcept {
8     return arr;
9 }
10
11 int main() {
12     int x[] {1,2,3,4};
13     std::vector c{1,2,3,4};
14
15     auto p1 = mybegin(c);
16     auto p2 = mybegin(x);
17 }
```

3.2.3 Iterator invalidation

반복자를 할당받은 후 컨테이너의 크기를 변경하면(resize) 어떻게 되는지 보자.

```

1 int main() {
2     int n = 0;
3     std::vector C{1,2,3,4};
4
5     auto it = c.begin();
6     n = *it;
7
8     c.resize(6); // 내부 버퍼 메모리 재할당
9     n = *it;    // error! invalid!
10 }
```

- 위 현상을 반복자 무효화(invalidation)라고 한다
- 컨테이너에서 반복자를 얻은 후 다양한 멤버 함수의 호출 결과로 인해 **내부 구조(메모리)가 변경된 경우 이전에 꺼내 놓은 반복자는 더 이상 사용할 수 없다.**
- 반복자는 언제 무효화 되는가? 컨테이너의 종류, 멤버 함수의 종류에 따라 다르다.
- 위 예시에서 vector의 크기를 줄이면 내부 구조가 변하지 않기 때문에 무효화되지 않는다. 크기를 키우는 경우 메모리를 재할당하면서 무효화된다!
- cppreference.com에 들어가 보면 컨테이너 별로 어떤 함수를 불렀을 때 무효화되는지 자세하게 나와있다

반복자의 크기와 함수 인자를 call by value로 작성하는 것

- 반복자는 크기가 크지 않고 복사 생성자 호출에 따른 오버헤드도 거의 없기 때문에 call by value로 함수 인자를 사용하는 것이 코딩 관례이다.

```

1 template<typename ITER, typename T>
2 ITER myfind(ITER first, ITER last, const T& value) { // 반복자는 call by value로 받는다
3     return first;
4 }
5
6 int main() {
7     std::list<int> l; std::cout << sizeof(l); // 8 byte
8     std::vector<int> v; std::cout << sizeof(v); // 8 byte
9     std::deque<int> d; std::cout << sizeof(d); // 24 byte
10
11    std::vector v{1,2,3,4};
12    auto ret = myfind(v.begin(), v.end(), 3);
13 }
```

컨테이너의 모든 요소를 화면에 출력하려면 range-for 구문을 사용하거나(가장 간결하고 좋은 방식) 반복자를 사용한다.

```

1 int main(){
2     std::vector v{1,2,3,4,5};
3
4     for(const auto& e : v) { // 1
5         std::cout << e << " ";
6     }
7 }
```

```
8 auto first = v.begin(); // 2
9 auto last = v.end();
10 while(first != last) {
11     std::print("{} , ", *first++);
12 }
13 std::println("");
14 }
```

3.2.4 std::ranges::begin

std::ranges::begin()은 C++20에서 새로 추가되었다. std::begin()과 std::ranges::begin()의 차이점에 대해 알아보자.

```
1 #include <vector>
2 #include <ranges> // std::ranges 네임스페이스 코드들이 구현되어 있음
3
4 int main(){
5     std::vector v{1,2,3,4,5};
6
7     auto it1 = std::begin(v);
8     auto it2 = std::ranges::begin(v);
9 }
```

std::begin(), std::end() 컨테이너 버전의 원리는 "컨테이너의 멤버 함수를 다시 호출"하는 것이다.

- 반환 타입이 "반복자(iterator)"가 아니라도 문제 없다.
- std::ranges::begin(), std::ranges::end()는 반환 타입이 반복자인지 검사한다.

```
1 class MyType {
2 public:
3     int begin() { return 0; }
4     int end() { return 0; }
5
6     int* begin() { return 0; }
7     int* end() { return 0; }
8 }
9
10 int main(){
11     MyType c;
12
13     auto it1 = std::begin(c);           // int, int* 타입 모두 작동한다
14     auto it2 = std::ranges::begin(c); // int 반환 타입인 경우 에러 발생한다
15 }
```

또 다른 예제를 보자.

```
1 int main(){
2     auto it1 = std::begin( std::vector{1,2,3} );           // 컨테이너에 임시 객체를 넘기는 경우
3
4     *it1 = 10;                                         // error. it1은 const iterator
5     int n = *it1;                                       // ok. 하지만 임시 객체는 파괴되었으므로 잘못된 주소를
6     가리키고 있다(dangling).
7
8     auto it2 = std::ranges::begin( std::vector{1,2,3} ); // rvalue인지 조사하는 과정에서 컴파일 에러가
9     발생한다
10 }
```

- std::ranges::begin(c)은 컴파일 과정에서 c가 borrowed ranges가 아니고 rvalue라면 컴파일 에러를 반환한다!

borrowed ranges

```
1 int main(){
2     std::string s{"to be or not to be"};
3 }
```

```

4 std::string ss = s; // 깊은 복사
5 std::string_view sv = s; // 복사하지 않고 s를 가리킴
6
7 auto it1 = std::ranges::begin( std::vector{1,2,3} ); // error
8 auto it2 = std::ranges::begin( std::string_view{s} ); // ok. it2는 s를 가리킨다
9 }
```

- `std::string`은 자원(문자열)을 소유한다.
- `std::string_view`는 자원을 소유하지 않고 다른 객체의 자원을 빌려서 사용한다 (대표적인 borrowed ranges)

c++20에서 추가된 용어와 개념

- Container: c++98부터 사용되던 용어
- View: c++20에서 추가된 용어. `std::string_view`의 개념을 vector, list 등으로 확장한 개념
- Range: c++20에서 추가된 용어. begin, end로 반복자를 꺼낼 수 있는 모든 타입을 말한다

3.2.5 Iterator value type

컨테이너의 구간 합을 구하는 코드를 작성해보자.

```

1 template<typename T>
2 auto sum(T first, T last) {
3     s = 0; // 이 곳의 타입을 어떻게 선언할 것인가?
4     while(first != last) { s += *first++; }
5     return s;
6 }
7
8 int main() {
9     std::vector v{1,2,3,4,5,6,7,8,9,10};
10
11    auto s = sum(v.begin(), v.end());
12
13    std::println("{}", s);
14 }
```

- T가 반복자일 때 T가 가르키는 요소의 타입은 어떻게 알 수 있을까?
- c++20: `std::iter_value_t<T>`
- c++98: `typename std::iterator_traits<T>::value_type`

반복자를 만들 때 `using value_type = T;`를 사용하여 value type을 명시해줘야 한다.

```

1 template<typename T> class vector_iterator {
2 public:
3     using value_type = T;
4 ...
5 };
```

```

1 template<typename T>
2 auto sum(T first, T last) {
3     typename T::value_type s = 0;
4     while(first != last) { s += *first++; }
5     return s;
6 }
7
8 int main() {
9     std::vector v{1,2,3,4,5};
10    int v[] {1,2,3,4,5};
11
12    auto s = sum(std::begin(v), std::end(v)); // v가 int*인 경우 typename T::value_type에서 에러가 발생한다!
13    std::println("{}", s);
14 }
```

-
- `typename T::value_type`는 에러가 발생하므로 `typename std::iterator_traits<T>::value_type`를 사용하여 컨테이너의 타입을 추론하였다.
 - C++20에서 `std::iter_value_t<T>`는 결국 `using iter_value_t = typename std::iterator_traits<T>::value_type`으로 축약한 것이다!

```

1 template<typename T> struct iterator_traits { // T가 컨테이너인 경우
2     using value_type = typename T::value_type;
3     ...
4 };
5
6 template<typename T> struct iterator_traits<T*> { // T가 포인터일 경우도 고려되어 설계되어 있다.
7     using value_type = T;
8     ...
9 };

```

3.2.6 Iterator category

아래와 같은 더블 링크드 리스트와 싱글 링크드 리스트 코드를 보자.

```

1 int main() {
2     std::list      ds{1,2,3,4}; // double linked list
3     std::forward_list ss{1,2,3,4}; // single linked list
4
5     auto p1 = ds.begin();
6     auto p2 = ss.begin();
7
8     ++p1; // ok
9     ++p2; // ok
10
11    --p1; // ok
12    --p2; // error. 자료구조 특징 상 존재할 수 없는 연산자
13 }

```

STL에서는 반복자가 할 수 있는 능력에 따라 6가지로 분류한다.

Category	Requirement	Container
output_iterator	write	
input_iterator	read, ++ (w/o multiple pass)	
forward_iterator	read, ++ (w/ multiple pass)	forward_list
bidirectional_iterator	read, ++, --	list
random_access_iterator	read, ++, --, +, -, []	
contiguous_iterator	read, ++, --, +, -, [], contiguous storage	vector
input_or_output_iterator	input_iterator output_iterator	

- 대표적인 질문이 "forward_iterator로는 write는 할 수 없나요?"인데 이는 안되는 것이 아니라 "[요구 조건이 아니라는 의미](#)"이다.

```

1 int main() {
2     std::list      ds{1,2,3,4};
3     std::forward_list ss{1,2,3,4};
4     std::vector     ve{1,2,3,4};
5
6     std::reverse( ds.begin(), ds.end() );
7     std::reverse( ss.begin(), ss.end() ); // error! reverse 연산자는 반드시 - 연산이 가능해야 한다
8     std::reverse( ve.begin(), ve.end() );
9 }

```

Iterator category가 왜 중요한가?

- STL의 다양한 알고리즘은 "[인자로 전달되는 반복자에 대한 요구 조건](#)"을 가지고 있다. 요구 조건은 카테고리로 제시되기 때문에 어떤 컨테이너가 어떤 카테고리에 속하는지 알아야 한다.

-
- STL 카테고리 관련 컴파일 에러는 매우 복잡하기 때문에 미리 이에 대한 지식을 알고 있으면 훨씬 수월하게 코딩할 수 있다.

Multiple passes란?

- `it1, it2`가 컨테이너의 같은 요소를 가르키는 반복자일 때 `*it1 == *it2`를 만족하고 `++it1, ++it2` 후에 다시 `*it1 == *it2`를 만족하는 것을 말한다
- "주어진 구간을 2개 이상의 반복자가 지나가도 동일한 값을 읽을 수 있는 것". 하나의 반복자로 읽기를 수행해도 요소의 값이 변하지 않는 것을 말한다
- `std::istream_iterator`는 입력 버퍼의 요소를 읽을 때 값을 꺼내기 때문에 multiple passes가 아니다.

특정 반복자가 category를 만족하는지 조사하려면 c++20의 Concept을 사용하면 된다.

```

1 template<typename T> void p() {
2     std::println("input: {0}, forward: {1}",
3     std::input_iterator<T>,
4     std::forward_iterator<T>);
5 }
6
7 int main() {
8     p<std::vector<int>::iterator>();
9     p<std::list<int>::iterator>();
10    p<std::istream_iterator<int>::iterator>();
11 }
```

`std::find()`와 `std::remove()` 함수를 보자.

```

1 int main() {
2     std::vector v{1,2,3,3,4,3};
3
4     auto it1 = std::find(v.begin(), v.end(), 4);
5     auto it2 = std::remove(v.begin(), v.end(), 3);
6 }
```

- `std::find`는 `value`를 찾을 때까지 `++연산`으로 이동만 하기 때문에 반복자가 multiple pass를 보장할 필요가 없다.
(입력 반복자가 `input iterator`이기만 하면 됨)

- `std::remove`는 실제 컨테이너의 항목을 제거하는 것이 아니라 `value`를 찾으면 뒤의 요소를 앞으로 복사(이동)
하는 연산을 수행한다.

- 따라서 `++연산`이 가능해야 하며 multiple pass를 보장해야 한다.

앞서 사용했던 `p()` 함수를 좀 더 발전시키면 반복자의 category를 정확하게 알 수 있다.

```

1 template<typename T>
2 void p(std::string_view name) {
3     if constexpr( std::contiguous_iterator<T> )
4         std::println("{:6} : contiguous_iterator", name);
5
6     else if constexpr( std::random_access_iterator<T> )
7         std::println("{:6} : random_access_iterator", name);
8
9     else if constexpr( std::bidirectional_iterator<T> )
10        std::println("{:6} : bidirectional_iterator", name);
11
12    else if constexpr( std::forward_iterator<T> )
13        std::println("{:6} : forward_iterator", name);
14
15    else if constexpr( std::input_iterator<T> )
16        std::println("{:6} : input_iterator", name);
17    }
18
19 int main() {
20     p<std::vector<int>::iterator>("vector");
21     p<std::list<int>::iterator>("list");
22     p<std::deque<int>::iterator>("deque");
```

```
23 p<std::array<int,5>::iterator>("array");
24 p<std::string::iterator>("string");
25 p<std::set<int>::iterator>("set");
26 p<std::map<int, int>::iterator>("map");
27 p<std::forward_list<int>::iterator>("forward_list");
28 }
```

3.2.7 Iterator operation

다음과 같이 벡터의 4번째 요소에 접근하는 코드를 작성해보자

```
1 void foo(auto it) {
2     it = it+3;
3     std::println("{}", *it); // 4 출력
4 }
5
6 int main() {
7     std::vector v{1,2,3,4,5,6,7,8,9,10}; // ok
8     std::list v{1,2,3,4,5,6,7,8,9,10}; // error!
9     foo(v.begin());
10 }
```

만약 vector가 아닌 list였다면 위 코드가 동작할까? list는 임의 접근반복자(random access iterator)가 아니기 때문에 `it = it+3` 코드가 동작하지 않는다.

- `++it, ++it, ++it`와 같이 입력할 수 있으나 보다 시피 코드가 비효율적이다.
- `foo`를 범용적인 함수로 만들기 위해 어떤 반복자가 들어와도 N 만큼 전진하는 코드를 작성해보자
- `std::advance(iterator, N)` 을 사용하면 iterator를 N번 전진한다. (vector, list 모두 동작!)

```
1 void foo(auto it) {
2     std::advance(it, 3); // vector, list 등 모든 컨테이너에 동작한다
3     std::println("{}", *it); // 4 출력
4 }
5
6 int main() {
7     std::vector v{1,2,3,4,5,6,7,8,9,10}; // ok
8     std::list v{1,2,3,4,5,6,7,8,9,10}; // ok
9     foo(v.begin());
10 }
```

`std::advance`와 유사한 몇 가지 함수에 대해 알아보자 (`std::prev, std::next, std::distance`)

```
1 int main() {
2     std::list s{1,2,3,4,5,6,7,8,9,10};
3
4     auto it1 = s.begin();
5
6     std::advance(it1, 3); // std::advance
7     std::println("{}", *it1);
8
9     auto it2 = std::prev(it1, 3); // std::prev
10    auto it3 = std::next(it1, 3); // std::next
11    std::println("{} , {} , {}", *it1, *it2, *it3);
12
13    auto n = std::distance(it2, it3);
14    std::println("{} , n", n); // std::distance
15 }
```

- `std::prev, std::next`는 반복자 자체는 변하지 않고 이동한 반복자를 반환한다
- `std::distance`는 두 반복자 사이의 거리를 반환한다

지금까지 배운 알고리즘들을 `c++` 버전 별로 정리해보자(`c++98, c++11, c++20`)

```

1 int main() {
2     std::list s{1,2,3,4,5,6,7,8,9,10};
3
4     auto first = s.begin();
5
6     // std::next
7     auto it1 = std::next(first);           // 한 칸 전진
8     auto it2 = std::next(first, 3);
9
10    // std::ranges::next
11    auto it3 = std::ranges::next(first);
12    auto it4 = std::ranges::next(first, 3);
13    auto it5 = std::ranges::next(first, s.end()); // 무조건 s.end()를 반환 (추후 counted iterator를 쓸 때
14                                // 유용하게 사용)
14    auto it6 = std::ranges::next(first, 3, s.end()); // 3칸 이동하거나 끝 중 먼저 도달하는 것을 반환
15    auto it7 = std::ranges::next(first, 20, s.end()); // 20칸 이동하거나 끝 중 먼저 도달하는 것을 반환
16
17    std::cout << "it1 == it4: " << (it1 == it4) << endl; // true
18    std::cout << "it5 == it7: " << (it5 == it7) << endl; // true
19 }

```

- c++98: std::advance, std::distance
- c++11: std::prev, std::next
- c++20: std::ranges::advance, std::ranges::next, std::ranges::prev, std::ranges::distance

3.2.8 ++ vs next

컨테이너에서 2번째 요소를 가리키는 반복자가 필요할 때 ++, next 중 어떤 것이 더 좋은 방법일까?

```

1 int main() {
2     std::vector c{1,2,3,4};
3
4     auto it1 = c.begin() + 1;          // #1
5     auto it2 = ++c.begin();          // #2
6     auto it3 = std::next(c.begin()); // #3
7     auto it4 = std::ranges::next(c.begin()); // #4
8 }

```

- c.begin() + 1은 반복자가 임의 접근 반복자인 경우만 가능하다. (list인 경우 예러 발생)
- auto it = ++c.begin()은 대부분 성공하지만 컴파일 에러가 발생할 수도 있다.
- 따라서 std::next, std::ranges::next를 사용하는 것을 권장한다!

++c.begin()이 예러가 발생하는 경우에는 어떤 것들이 있을까?

```

1 int next(int addr) { return ++addr; }
2 int x[5]{1,2,3,4,5};
3 int* address_of_x() { return &x[0]; }
4
5 int main() {
6     int* p1 = ++( &x[0] );          // error
7     int* p2 = ++( address_of_x() ); // error
8
9     int* addr = &x[0];
10    int* p3 = ++addr;              // ok. addr에 담기면서 lvalue가 되므로 컴파일된다
11    int* p4 = next( &x[0] );       // ok
12
13    std::cout << *p1 << ", " << *p2 << endl;
14    std::cout << *p3 << ", " << *p4 << endl;
}

```

- ++(&variable): 연산자가 반환하는 주소 같은 "rvalue"이다. rvalue는 ++ 연산을 할 수 없다 (compile error!)

임의의 유저 타입에 대한 반복자 코드를 살펴보자.

```

1 class myiterator {

```

```

2 int* current;
3 public:
4 myiterator(int* p = nullptr) : current{p} {}
5
6 myiterator& operator++() { ++current; return *this; }
7 int& operator*() { return *current; }
8 };
9
10 int x[5]{1,2,3,4,5};
11
12 int* get_raw_pointer() { return &x[0]; }
13 myiterator get_myiterator() { return &x[0]; }
14
15 int main() {
16 auto it1 = ++get_raw_pointer(); // error!
17 auto it2 = ++get_myiterator(); // ok
18 std::println("{}", *it2);
19 }
```

- 왜 굳이 raw pointer와 동일한 동작을 하는(++, *) 반복자 클래스를 작성했을까?
- ++(get_raw_pointer())의 반환값은 "주소 값, rvalue"이다. 따라서 ++ 연산을 할 수 없다. (**compile error!**)
- ++(get_myiterator())의 반환값 또한 "myiterator 타입의 임시객체, rvalue"이다. **하지만 rvalue는 동호 왼쪽에 놓일 수는 있지만 멤버 함수를 호출할 수 있다.** 즉, ++연산은 내부적으로 () .operator++() 함수를 호출하게 된다.

```

1 class myiterator {
2 int* current;
3 public:
4 myiterator(int* p = nullptr) : current{p} {}
5
6 myiterator& operator++() { ++current; return *this; }
7 int& operator*() { return *current; }
8 };
9
10 class MyContainer {
11 int data[5]{1,2,3,4,5};
12 public:
13 using iterator = myiterator; // 정상 동작한다. 만약 int*로 되어 있었다면 ++.mc.begin()에서 에러가 발생한다.
14
15 iterator begin() { return data; }
16 iterator end() { return data+5; }
17 };
18
19 int main() {
20 MyContainer mc;
21 auto it = ++mc.begin(); // ok
22 std::println("{}", *it);
23 }
```

3.2.9 std::copy vs std::ranges::copy

컨테이너 c1의 모든 요소를 c2에 복사하고 싶다고 가정하자.

```

1 int main() {
2 std::vector c1{1,2,3,4,5};
3 std::vector c2{0,0,0,0,0,0,0};
4
5 c2 = c1; // #1
6
7 c2.assign(c1.begin(), c1.end()); // #2
8
9 for( int i=0; i<c1.size(); i++ ) // #3
10 c2[i] = c1[i];
11 }
```

```
12 std::copy(c1.begin(), c1.end(), c2.begin());  
13 }
```

- **대입연산자 사용** (`c2 = c1`): `c1`, `c2`가 동일 컨테이너여야 한다. `c2`의 기존 요소가 완전히 제거되고 `c1`을 복사한다.
- **assign 멤버 함수 사용**: `c1`, `c2`가 다른 컨테이너여도 사용 가능하다. `c2`의 기존 요소가 완전히 제거되고 `c1`을 복사한다.
- **반복문과 [] 연산자**: `c1`, `c2`가 `std::list`라면 [] 연산자 지원이 안된다. `c2`에서 `c1`의 크기만큼만 요소가 복사된다.
- **`std::copy`, `std::ranges::copy`**: 위 방법과 동일한 복사가 되고 `std::list` 포함한 대부분의 컨테이너에서 실행 가능하다.

`std::copy - 128) * 64 + (' - 128) 뺀 std::ranges::copy 알고리즘은 무엇이 다를까? std::copy (c++98): 인자로 반복자만 전달이 가능하다. 반환값은 출력 컨테이너의 반복자이다. std::ranges::copy (c++20): 반복자 버전과 컨테이너 버전을 제공한다. 반환 값은 in_out_result이다.`

```
1 int main() {  
2     std::vector c1{1,2,3,4,5};  
3     std::vector c2{0,0,0,0,0,0,0};  
4  
5     std::copy(c1.begin(), c1.end(), c2.begin());           // 반복자만 전달 가능  
6  
7     std::ranges::copy(c1.begin(), c1.end(), c2.begin()); // 반복자 뿐만 아니라  
8     std::ranges::copy(c1, c2.begin());                      // 컨테이너도 전달 가능  
9 }
```

두 알고리즘은 반환값도 서로 다르다.

```
1 int main() {  
2     std::vector c1{1,3,5,7,9};  
3     std::vector c2{2,4,6,8,10};  
4  
5     auto ret1 = std::copy(c1.begin(), std::next(c1.begin(),3), c2.begin()); // c2.end() 값을 반환한다  
6     std::println("{}, {}", *ret1);  
7  
8     auto ret2 = std::ranges::copy(c1.begin(), std::next(c1.begin(),3), c2.begin()); // (in and out)  
9     ret2.in은 c1.end()를 반환하고 ret2.out는 c2.end()를 반환한다  
10    std::println("{}, {}", *(ret2.in), *(ret2.out));  
11 }
```

3.2.10 Reverse iterator

반대 방향으로 이동하는 반복자인 reverse iterator에 대해 알아보자

```
1 int main() {  
2     std::vector v{1,2,3,4,5,6,7,8,9,10};  
3  
4     auto it = std::next(v.begin(), 5); // 6  
5  
6     std::reverse_iterator ri{ it };  
7  
8     std::println("{} , {}", *it, *ri); // 6, 5  
9     it++; ri++;  
10    std::println("{} , {}", *it, *ri); // 7, 4  
11 }
```

- `ri++`를 수행하면 한 칸 뒤로 이동한다

reverse iterator는 기존의 방향을 반대로 해주는 기능을 가진 "iterator adaptor"이다.

- `++ri == --it`
- `--ri == ++it`
- `*ri = *std::prev(it)`: 현재 가리키는 곳에서 한 칸 뒤에 값을 반환한다.
- 양방향 반복자(bidirectional iterator) 조건을 만족하는 반복자에만 사용 가능하다.

reverse iterator 객체를 생성하는 방법은 두 가지가 있다. (#include <iterator> 헤더 필요)

```
1 #include <iterator>
2
3 int main() {
4     std::vector v{1,2,3,4,5,6,7,8,9,10};
5
6     auto it = std::next(v.begin(), 5);
7
8     std::reverse_iterator ri{ it }; // #1
9     std::reverse_iterator<std::vector<int>::iterator> ri2{ it }; // c++17 이전에는 타입을 명시해줘야 한다.
10
11    auto ri3 = std::make_reverse_iterator(it); // #2
12 }
```

또한 reverse iterator에 접근하기 위해서는 rbegin, rend를 사용해야 한다.

```
1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     auto first = v.begin();
5     auto last = v.end();
6
7     auto rfirst = v.rbegin(); // std::make_reverse_iterator(v.end())와 동일
8     auto rend = v.rend(); // std::make_reverse_iterator(v.begin())와 동일
9
10    std::vector<int>::reverse_iterator ri = v.begin();
11 }
```

- reverse iterator는 컨테이너에서 {name}::reverse_iterator로 바로 사용할 수 있다.

const iterator는 값을 변경할 수 없는 반복자를 말한다. (상수 버전 포인터와 동일). reverse iterator 또한 const 버전이 존재한다.

```
1 int main() {
2     std::vector v{1,2,3,4,5};
3
4     std::vector<int>::iterator it1 = v.begin();
5     std::vector<int>::reverse_iterator it1 = v.rbegin();
6     std::vector<int>::const_iterator it1 = v.cbegin();
7     std::vector<int>::const_reverse_iterator it1 = v.crbegin();
8
9     std::find(it1, it2, 3); // error! 동일 타입이 아니므로 에러 발생
10 }
```

- const 타입 또한 멤버 함수가 아닌 일반 함수 버전과 std::ranges 버전도 모두 제공된다.

3.2.11 Insert iterator

선형 컨테이너 끝에 요소를 추가하는 2가지 방법이 있다

- 컨테이너의 push_back() 멤버 함수를 사용하는 방법
- 후방 삽입 반복자 (std::back_insert_iterator)를 사용하는 방법

```
1 int main() {
2     std::list s{1,2,3};
3
4     s.push_back(10); // #1
5
6     std::back_insert_iterator bi{s}; // #2
7     *bi = 20; // s.push_back(20)과 동일
8     *bi = 30; // s.push_back(30)과 동일
9 }
```

-
- `std::back_insert_iterator`와 같은 반복자를 **삽입 반복자(insert iterator)**라고 한다 (c++98부터 지원)
 - 전방, 후방, 임의 삽입 등 3가지 형태로 제공된다.

삽입 반복자를 생성하는 방법에 대해 알아보자.

```
1 int main() {
2     std::list s{1,2,3};
3
4     std::back_insert_iterator bi1{s};           // c++17
5     std::back_insert_iterator<std::list<int>> bi2{s}; // c++17 이전에는 타입을 직접 명시해줘야 한다
6
7     auto bi3 = std::back_inserter(s);           // 편의 함수 템플릿(convienient function template)을
8         사용한다. (c++98부터 지원)
9
10    *bi1 = 10;
11    *bi2 = 20;
12    *bi3 = 30;
13 }
```

삽입 반복자와 copy 알고리즘에 대해 알아보자

- 컨테이너의 모든 요소를 다른 컨테이너 끝에 추가하고 싶다면
- 반복문과 `push_back()` 멤버 함수를 이용하는 방법과
- `copy` 알고리즘과 삽입 반복자를 이용하는 방법이 있다.

아래와 같이 벡터의 모든 요소를 리스트의 끝에 추가하고 싶은 경우를 보자.

```
1 int main() {
2     std::vector v{1,2,3,4,5};
3     std::list s1{0,0,0,0,0};
4     std::list s2{0,0,0,0,0};
5
6     for(auto e : v)                      // #1 push_back 사용
7         s1.push_back(e);
8
9     std::back_insert_iterator bi{s2};       // #2 copy 알고리즘과 후방 삽입 반복자 이용
10    std::ranges::copy(v, bi);
11
12    std::ranges::copy(v, std::back_inserter(s2));
13 }
```

(주의) `copy` 사용 시 컨테이너의 요소를 덮어 쓰는 것(overwrite)과 요소를 삽입(insert)하는 것을 잘 구별하고 사용해야 한다!

```
1 int main() {
2     std::vector v{1,2,3,4,5};
3     std::list s1{0,0,0,0,0};
4     std::list s2{0,0,0,0,0};
5
6     copy(v, s1.begin());                // 덮어쓰기(overwrite)
7     copy(v, std::back_inserter(s2)); // 삽입(insert)
8
9     std::list<int> s3;
10    copy(v, s3.begin());            // error! 요소가 없는 컨테이너에 복사하려고 한다.
11    copy(v, std::back_inserter(s3)); // ok! s3부터 값이 추가된
12 }
```

다음으로 삽입 반복자의 3가지 종류에 대해 알아보자.

- 임의 삽입 반복자는 생성자 인자로 2개를 전달해야 한다.
- `std::insert_iterator ii(container, iterator);`
- `auto ii = std::inserter(container, iterator);`
- iterator 앞쪽에 값을 넣겠다는 의미이다.

종류	형태
전방 삽입	std::front_insert_iterator<> std::front_inserter
후방 삽입	std::back_insert_iterator<> std::back_inserter
임의 삽입	std::insert_iterator<> std::inserter

```

1 int main() {
2     std::list s{1,2,3,4,5};
3
4     // #1 클래스 템플릿 이용 (c++17 )
5     std::front_insert_iterator fi1{s};
6     std::back_insert_iterator bi1{s};
7     std::insert_iterator ii1{s, std::next(s.begin(), 2)}; // 3을 가리킴. 삽입을 수행하면 2와 3 사이에 삽입됨
8
9     // c++17 이전에는 타입을 명시해줘야 한다
10    std::front_insert_iterator<std::list<int>> fi2{s};
11    std::back_insert_iterator<std::list<int>> bi2{s};
12    std::insert_iterator<std::list<int>> ii2{s, std::next(s.begin(), 2)};
13
14    // #2 편의 함수 템플릿(convinent function template)을 사용한다
15    auto fi3 = std::front_inserter(s);
16    auto bi3 = std::back_inserter(s);
17    auto ii3 = std::inserter(s, std::next(s.begin(), 2));
18
19    *fi3 = 10;
20    *bi3 = 20;
21    *ii3 = 30;
22
23    show(s); // 10, 1, 2, 30, 3, 4, 5, 20 - 128) * 64 + (' - 128) 뿐
24        - 128) * 64 + (' - 128) 끝 - 128) * 64 + (' - 128) 끝.
}

```

전방 삽입 반복자와 임의 삽입 반복자의 차이점에 대해 알아보자

```

1 int main() {
2     std::vector v{1,2,3,4,5};
3     std::list s1{0,0,0,0,0};
4     std::list s2{0,0,0,0,0};
5
6     std::ranges::copy(v, std::front_inserter(s1));
7     std::ranges::copy(v, std::inserter(s2, s2.begin()));
8
9     show(s1); // 5,4,3,2,1,0,0,0,0,0
10    show(s2); // 1,2,3,4,5,0,0,0,0,0
11 }

```

- 전방 삽입 반복자를 사용하면 기존 값들이 **거꾸로** 들어간다. 하지만 임의 삽입 반복자는 **제대로** 들어간다.

다음으로 삽입 반복자를 사용할 때 주의 사항에 대해 살펴보자.

```

1 int main() {
2     std::vector v{1,2,3,4,5};
3
4     auto p1 = std::front_inserter(v); // error!
5     auto p2 = std::inserter(v, v.begin()); // ok
6     *p1 = 10;
7     *p2 = 10;
8
9     ++p2; // ok. ++연산자는 아무일도 하지 않는다
10
11 using T = std::back_insert_iterator<std::vector<int>>;

```

```
12 std::println("{0}", std::output_iterator<T, int>);  
13 }
```

-
- `std::vector`는 전방 삽입 반복자를 사용할 수 있다.(`push_front`가 없음!)
 - 모든 반복자는 `++ 연산`을 제공해야 한다. 단 삽입 반복자의 `++`은 아무 일도 하지 않는다.

삽입 반복자의 category는 **출력 반복자(output iterator)**로 분류된다.

3.2.12 `std::counted_iterator & std::default_sentinel`

`counted iterator`라는 개념은 c++20에서 추가되었으며 `<iterator>` 헤더에 포함되어 있다.

- `counted iterator`는 기존의 반복자에 추가적으로 "갯수를 관리하는 기능을 추가"한 반복자 어댑터이다.
- 동작 방식은 기존 반복자와 완전히 동일하다.

```
1 int main() {  
2     std::vector v{1,2,3,4,5,6,7,8,9,10};  
3  
4     auto it = v.begin();  
5  
6     std::counted_iterator ci{it, 5}; // it부터 5개까지만 접근할 수 있다는 의미  
7  
8     ++it;  
9     ++ci;  
10  
11    std::println("{0}, {1}", *it, *ci); // 2, 2  
12  
13    std::println("{0}", ci.count()); // 4 (++ci에서 1 감소)  
14  
15    while(ci.count() != 0) {  
16        std::println("{0}", *ci++); // 2,3,4,5  
17    }  
18  
19    ++ci;  
20    std::println("{0}", ci.count()); // -1;  
21    std::println("{0}", *ci); // 이론적으로는 undefined이지만 실제 구현은 return *current가 된다. (사용  
22    안할 것을 권장)  
}
```

-
- `++ci` 연산 시 `++current`, `--count`가 내부적으로 행된다.
 - `count < 0` 인 경우 `*ci`를 하면 어떻게 될까? `count`는 계속 줄어들고 `current`는 `undefined`가 된다.

`counted iterator`를 다양한 알고리즘에 전달하려면 어떻게 해야 할까?

- 알고리즘에 반복자를 전달하려면 [first, last) 반복자 쌍이 필요하다. ex) `auto ret = std::find(ci, ???, 3);`
- `counted iterator`는 "한 개의 반복자 안에 범위의 끝에 대한 정보도 포함"하고 있다
- 하지만 std 기본 알고리즘은 반복자 쌍을 요구하므로 이 때 사용하는 것이 **default sentinel**이라는 객체이다.

```
1 int main() {  
2     std::vector v{1,2,3,4,5,6,7,8,9,10};  
3  
4     auto it = v.begin();  
5  
6     std::counted_iterator ci{it, 5};  
7  
8     // while(ci.count() != 0) // 이렇게 사용해도 되지만  
9     while(ci != std::default_sentinel) { // default sentinel을 써도 된다.  
10         std::println("{0}", *ci++);  
11     }  
12 }
```

default sentinel 구조체는 다음과 같이 정의되어 있다.

```

1 struct default_sentinel_t {};
2
3 inline constexpr default_sentinel_t default_sentinel{};
4
5 constexpr bool
6 operator==(const std::counted_iterator& ci,
7 std::default_sentinel_t) noexcept {
8 return ci.count == 0;
9 }

```

- == 연산자가 호출되면 ci.count가 0인지 검사한다.

c++98 시절에는 sentinel이라는 개념이 없었기 때문에 STL find 알고리즘이 다음과 같이 설계되어 있다.

```

1 template<class InputIt, class T>
2 InputIt find(InputIt first, InputIt last, const T& value);

```

- 구간을 나타내는 **반복자 쌍이 반드시 같은 타입**이어야 한다.

c++20에서 추가된 constrained 알고리즘 버전의 find는 다음과 같이 구현되어 있다.

```

1 template<std::input_iterator I,
2 std::sentinel_for<I> S,
3 class T, class Proj = std::identity >
4 requires
5 std::indirect_binary_predicate<ranges::equal_to,
6 std::projected<I, Proj>, const T*>
7
8 constexpr I find(I first, S last,
9 const T& value, Proj proj = {});

```

- 핵심은 구간을 나타내는 **반복자 쌍이 다른 타입도 가능하다**는 것이다.

```

1 int main() {
2 std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4 std::counted_iterator ci{v.begin(), 5};
5
6 auto ret = std::find(ci, std::default_sentinel, 3);      // error! 반복자 쌍이 반드시 같은 타입이어야 한다
7 auto ret = std::ranges::find(ci, std::default_sentinel, 3); // ok.
8
9 std::println("{}", *ret);
}

```

std::counted_iterator와 std::default_sentinel을 legacy function(c++98)에 전달하려면 어떻게 해야 할까?

- iterator와 sentinel의 "공통의 타입"을 만들어야 한다.
- **std::common_iterator**: iterator와 sentinel을 모두 담을 수 있는 공통의 타입을 설계할 때 사용한다.

```

1 int main() {
2 std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4 std::counted_iterator ci{v.begin(), 5};
5
6 using T = std::common_iterator<                                // 두 타입의 공통의 타입 선언
7 std::counted_iterator<std::vector<int>::iterator>,
8 std::default_sentinel_t>;
9
10 auto ret = std::find( T{ci}, T{std::default_sentinel}, 3);
11
12 std::println("{}", *ret);
}

```

3.2.13 Iterator and sentinel

find 알고리즘을 사용해서 선형 검색을 수행하려면 검색 대상 구간을 인자로 전달해야 한다.

c++98 std::find	시작과 끝(past the last element)를 나타내는 "2개의 반복자"를 전달한다. 반드시 같은 타입이어야 한다. InputIt find(InputIt first, InputIt last, const T& value);
c++20 std::ranges::find	시작을 나타내는 반복자와 끝을 나타내는 sentinel을 전달한다. sentinel은 반복자일 수도 있고 다른 형태의 객체일 수도 있다. I find(I first, S last, ...);

sentinel이라는 개념은 c++20에서 처음 제안된 개념으로 보통 반복자 구간의 끝을 나타낼 때 사용한다

- 사전적 의미1: 보초, 파수병
- 사전적 의미2: (특정 정보 블럭의 시작과 끝을 나타내는) 표지
- std::default_sentinel_t를 사용하려면 그 자체로는 "empty class"이므로 어떠한 정보도 담고 있지 않다
- 1번째 인자로 전달되는 iterator 자체에 구간 끝에 대한 정보가 있어야 한다.

```

1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     auto it = v.begin();
5     std::counted_iterator ci{it, 5};
6
7     auto ret1 = std::ranges::find(ci, std::default_sentinel, 3); // ok
8     auto ret2 = std::ranges::find(it, std::default_sentinel, 3); // error! it에 구간 끝에 대한 정보가 없다
9     auto ret3 = std::ranges::find(it, v.end(), 3);           // ok
10 }
```

std::sentinel_for<S, I>: S가 I의 sentinel로 사용 가능한지를 조사할 때 사용하는 concept이다.

```

1 template<class S, class I>
2 concept sentinel_for =
3     std::semiregular<S> &&
4     std::input_or_output_iterator<I> &&
5     _WeaklyEqualityComparableWith<S, I>;
```

```

1 int main() {
2     using vi_t = std::vector<int>::iterator;
3     using ci_t = std::counted_iterator<vi_t>;
4
5     bool b1 = std::sentinel_for<std::default_sentinel_t, ci_t>; // true. default_sentinel_t를 벡터의 ci로
6     쓸 수 있다
7     bool b2 = std::sentinel_for<std::default_sentinel_t, vi_t>; // false. default_sentinel_t를 벡터의
8     반복자로 쓸 수 없다
9
10    std::println("{} {}", b1, b2);
11
12    bool b3 = std::sentinel_for<vi_t, vi_t>; // true
13 }
```

따라서 최신 STL find 알고리즘에는 sentinel을 고려하여 코드가 구현되어 있다.

```

1 template<typename InputIter>
2 void find_cpp98(InputIter first, InputIter last) {}
3
4 template<std::input_or_output_iterator I, std::sentinel_for<I> S>           // concept을 사용하여 구현되어
5     void find_cpp20(I first, S last) {}
```

3.2.14 Ostream iterator

반복자를 사용하여 화면에 숫자/문자열을 출력할 수 있다

```
1 int main() {
2     int n = 10;
3
4     std::cout << n << std::endl;           // #1
5
6     std::ostream_iterator<int> p(cout, ", "); // #2 반복자를 통한 출력
7     *p = 20;                                // cout << 20 << ", "과 동일
8     *p = 30;                                // cout << 30 << ", "과 동일
9
10    std::list<int> s = {1,2,3};
11    std::copy(std::begin(s), std::end(s), p); // 리스트에 있는 모든 내용을 p로 복사하는데 p가 출력 반복자이므로
12        // 화면에 출력됨
13    std::fill_n(p, 3, 0);                   // 출력 반복자에 0을 세 번 출력한다
14
15    ++p;                                  // ok. 아무 동작도 하지 않는다
}
```

- 스트림 반복자는 입출력 스트림에서 요소를 읽거나 쓰기 위한 반복자이다.
- **ostream_iterator**: 출력 스트림을 사용해서 출력을 하는 반복자로 copy 등의 알고리즘 함수를 사용해서 스트림에 출력할 때 사용한다
- delimiter(", ")를 추가해도 되고 없이 사용해도 된다.
- 삽입 반복자(insert iterator)와 마찬가지로 ++ 연산은 구현이 되어 있지만 아무 동작도 하지 않는다.

파일에 입출력을 하고 싶을 경우 다음과 같이 사용한다

```
1 int main() {
2     ofstream f("a.txt");
3
4     std::ostream_iterator<int> p(f, ", "); // 파일에 출력하는 반복자
5
6     *p = 20;
7     *p = 30;                            // 결과 값이 파일에 출력(작성)된다.
8 }
```

3.2.15 Ostreambuf iterator

ostream 반복자와 구분되는 ostreambuf iterator에 대해 알아보자

```
1 int main() {
2     std::ostreambuf_iterator<char> p(cout); // char 타입만 받고 delimiter 입력이 불가능하다 (wchar_t)
3     *p = 65;                                // 'A'가 출력된다
4 }
```

- ostreambuf 반복자는 <char>만 출력할 수 있으며 delimiter 입력이 불가능하다.
- ostreambuf에 대해 제대로 이해하려면 cout에 대한 깊은 이해가 먼저 필요하다

cout << 65 코드가 실행되면

- 출력 버퍼에 '6', '5' 문자를 넣는다 (ostream 클래스)
- 버퍼를 관리하는 streambuf 클래스에서 버퍼가 가득 차면 화면에 출력한다
- cout.rdbuf() 함수를 사용하면 streambuf의 포인터를 얻을 수 있다.

```
1 int main() {
2     std::cout << 65;
3
4     streambuf* buf = cout.rdbuf();
5     buf->sputc(65);                  // 아스키코드 A(65)가 출력된다
6 }
```

-
- ostream_iterator: stream(cout)을 사용해서 화면을 출력하는 반복자 (서식화된 출력)
 - ostreambuf_iterator: streambuf를 사용해서 화면을 출력하는 반복자 (문자 타입만 출력)
 - 문자열만 사용하는 경우 ostreambuf_iterator를 사용하는 것이 속도가 조금 더 빠르다
-

```
1 int main() {
2     std::ostream_iterator<int> p1(cout, ", ");
3     *p1 = 10;
4
5     std::ostream_iterator<char> p2(cout.rdbuf()); // 아스키코드 출력
6     *p2 = 'A';
7
8     std::ostream_iterator<wchar_t> p3(wcout.rdbuf()); // 유니코드 출력
9     *p3 = L'A';
10 }
```

깊은 이해를 위해 ostream iterator의 구현 코드를 살펴보자.

```
1 template<typename T, typename CharT = char, typename Traits = char_traits<CharT>> class
2     eostream_iterator {
3     std::basic_ostream<CharT, Traits>* stream;
4     const CharT* delimiter;
5
6     public:
7     using iterator_category = output_iterator_tag;
8     using value_type = void;
9     using pointer = void;
10    using reference = void;
11    using difference_type = void;
12
13    using char_type = CharT;
14    using traits_type = Traits;
15    using ostream_type = std::basic_ostream<CharT, Traits>;
16
17    eostream_iterator(ostream& os, const CharT* const deli=0) : stream(&os), delimiter(deli) {}
18
19    eostream_iterator& operator*() { return *this; }
20    eostream_iterator& operator++() { return *this; }
21    eostream_iterator& operator++(int) { return *this; }
22
23    eostream_iterator& operator=(const T& v) {
24        *stream << v;
25        if(deli != 0)
26            *stream << delimiter;
27        return *this;
28    };
29
30    int main() {
31        eostream_iterator<int> p(cout, ", ");
32        *p = 10; // ( p.operator*() ).operator=(10) 호출
33    }
```

3.2.16 Istream iterator

출력 반복자와 반대로 값을 입력하는 istream iterator 또한 존재한다

```
1 int main() {
2     std::istream_iterator<int> p1(cin);
3     int n = *p1; // cin이 실행되어 값을 입력받는다
4
5     std::cout << n << std::endl;
6 }
```

ostreambuf iterator와 동일하게 istreambuf iterator 또한 존재한다. istream_iterator는 white space를 입력받지 못하지만 istreambuf_iterator는 white space를 입력받을 수 있다.

스트림 반복자	출력 대상	출력 형태
ostream_iterator	basic_ostream	서식화된 출력
ostreambuf_iterator	basic_ostreambuf	CharT 출력
istream_iterator	basic_istream	서식화된 입력
istreambuf_iterator	basic_istreambuf	CharT 입력

파일로부터 입력 반복자를 받아서 출력 반복자를 통해 화면에 출력하는 코드를 작성해보자

```
1 int main() {
2     std::ifstream f("a.txt");
3
4     std::istreambuf_iterator<char> p1(f), p2; // 디폴트 생성자 p2는 end of stream을 나타낸다.
5     std::ostream_iterator<char> p3(cout);
6
7     std::copy(p1, p2, p3); // p1부터 p2(end of stream)까지 p3 반복자에 복사한다. 즉 모든 파일
8     내용이 출력된다.
}
```

3.3 Algorithm

3.3.1 Erase-remove idioms

다음과 같이 벡터에서 3을 지우는 코드를 살펴보자

```
1 int main() {
2     std::vector v{1,2,3,1,2,3,1,2,3,1};
3
4     auto ret = std::remove(v.begin(), v.end(), 3);
5     show(v); // 1,2,1,2,1,2,3,1 이 나온다. 마지막 3이 지워지지 않았다.
6 }
```

- 마지막 원소 3은 왜 지워지지 않았을까?

std::remove는 다음과 같이 구현되어 있다.

```
1 template<typename ForwardIt, typename T>
2 ForwardIt remove(ForwardIt first,
3 ForwardIt last,
4 const T& value)
5 {
6     // [first, last) 구간에서 value를 제거
7 }
```

- remove 구현은 대부분의 컨테이너에서 동작할 수 있도록 코드가 작성되어 있다.
- remove 알고리즘은 내부적으로 반복자만 알 수 있고, 컨테이너는 알 수 없으므로 컨테이너의 크기 자체를 변경하지 않는다. (알고리즘은 컨테이너를 알지 못한다.)
- remove의 구현 원리는 조건을 만족하는 요소를 찾으면 뒤에 있는 요소를 현재 위치로 이동(move)한다. 즉 [1,2,1,2,1,2,1] 다음에 기존에 있던 [2,3,1]도 그대로 존재한다.

```
1 int main() {
2     std::vector v{1,2,3,1,2,3,1,2,3,1};
3
4     auto ret = std::remove(v.begin(), v.end(), 3);
5     show(v);
6
7     v.erase(ret, v.end()); // ret을 기준으로 뒷 부분을 제거하면
8     show(v); // 1,2,1,2,1이 제대로 출력된다.
9 }
```

-
- 한 줄로 줄여서 `v.erase(std::remove(v.begin(), v.end(), 3), v.end());` 로 쓰기도 한다. (**erase-remove 기술**)
 - 연속된 메모리를 사용하는 vector에서는 가장 효율적인 방법이다.

`std::remove`과 `std::ranges::remove`의 차이점은 무엇일까?

```

1 template<typename ForwardIt, typename T>
2 ForwardIt remove(ForwardIt first,
3 ForwardIt last,
4 const T& value)
5 {
6     first = std::find(first, last, value);
7     if(first != last)
8         for(ForwardIt i = first, ++i != last; )
9             if(!(*i == value))
10                 *first++ = std::move(*i);    // std::move가 사용된다
11     return first;
12 }
```

- `std::remove()`는 `std::move()`를 사용하여 뒤에 있는 요소를 앞으로 이동(move)시킨다.

```

1 int main(){
2     std::vector v{1,3,1,3,1,3,10,3,9,3};
3
4     auto ret1 = std::remove(v1.begin(),
5                             std::next(v1.begin(), 7), 3); // 끝까지 지우는 것이 아닌 구간 내에서 3을 지우는 코드를 보자
6
7     show(v1); // 1,1,1,10,[1,3,10],3,9,30] 출력된다. 중간에 [1,3,10]이 잘
8     지워지지 않았다.
9
10    v1.erase(ret1, v1.end()); // 1,1,1,10만 출력된다. 하지만 우리가 원하는 것은 [1,3,10]을
11        제외한 값을 얻고 싶다
12    show(v1);
13
14    v1.erase(ret1, std::next(v1.begin(), 7)); // 1,1,1,10,3,9,30] 출력된다
15    show(v1);
16 }
```

- 결국 구간 내에서 제대로 remove-erase idiom을 사용하려면 `v1.erase(et1, std::next(v1.begin(), 7));`와 같이 제거해줘야 한다.
- `std::ranges::remove`를 사용하면 반환값이 `std::ranges::subrange`인데 여기에 v1의 잘라진 구간의 처음과 끝 정보가 포함되어 있다.

```

1 int main(){
2     std::vector v{1,3,1,3,1,3,10,3,9,3};
3
4     auto ret1 = std::ranges::remove(v1.begin(),
5                                     std::next(v1.begin(), 7), 3);
6     v1.erase(ret1.begin(), ret1.end()); // ret1이 subrange이므로 이 구간을 지우면
7                                     // 알아서 중간 부분이 지워진다.
8     show(v1); // 1,1,1,10,3,9,3
9 }
```

3.3.2 Algorithm vs member function

알고리즘의 `remove`와 멤버 함수의 `remove`가 있을 때 어떤 것을 사용하면 좋을까?

```

1 int main(){
2     std::list c{1,2,3,1,2,3};
3
4     c.erase(std::remove(c.begin(), c.end(), 3), c.end()); // 알고리즘 remove 사용
5     c.remove(3); // 멤버함수 remove 사용
6 }
```

-
- `std::list`는 별도의 최적으로 구현된 `remove` 멤버 함수가 존재한다.
 - 컨테이너에 알고리즘과 동일한 이름의 멤버 함수가 있다면 멤버 함수를 사용해라.

왜 컨테이너 안에 알고리즘과 동일한 이름의 멤버 함수가 있을까?

- 이유 1) 컨테이너의 반복자를 "알고리즘으로 보낼 수 없을 때"
- `std::sort` 알고리즘은 introsort(quick + heap) 기법을 사용하는데 "random access iterator"를 인자로 요구한다. 따라서 `list`의 반복자를 전달할 수 없다.
- `std::list` 안에는 다른 방식으로 구현된 `sort` 함수가 존재한다.

```
1 int main(){
2     std::list c{1,2,3,1,2,3};
3
4     // #1
5     std::sort(s.begin(), s.end());           // error
6     std::ranges::sort(s.begin(), s.end());   // error
7     std::ranges::sort(s);                  // error
8     s.sort();                           // ok
9 }
```

- 이유 2) 컨테이너의 "멤버 함수가 알고리즘보다 효율적일 때"
- `list`의 반복자도 `std::remove`에 전달할 수 있지만 `std::list`의 `remove`를 사용하는 것이 효율적이다.

```
1 int main(){
2     std::list c{1,2,3,1,2,3};
3
4     // #2
5     c.erase(std::remove(c.begin(), c.end(), 3), c.end()); // 알고리즘 remove 사용
6     c.remove(3);                                         // 멤버함수 remove 사용
7 }
```

컨테이너 안에 있는 모든 요소를 뒤집고 싶다!

- 1. 멤버 함수 중에서 `reverse()`가 있으면 사용
- 2. 멤버 함수가 없으면 알고리즘 `reverse()` 사용

```
1 int main(){
2     std::list c{1,2,3,4,5};
3     std::vector v{1,2,3,4,5};
4
5     std::ranges::reverse(v); // list, vector 모두 사용 가능
6     s.reverse();           // list에만 존재 (권장)
7 }
```

3.3.3 `std::erase`, `std::erase_if`

erase-remove idioms를 사용하면 컨테이너에서 원하는 요소를 지울 수 있지만 코드가 장황해보이는 단점이 있다. C++20부터는 `std::erase`, `std::erase_if` 알고리즘을 제공하여 편하게 원하는 요소를 제거할 수 있다.

```
1 int main(){
2     std::list s{1,2,3,4,5};
3     std::vector v{1,2,3,4,5};
4
5     auto cnt1 = std::erase(v, 3); // C++20부터 지원
6     auto cnt2 = std::erase(s, 3); // C++20부터 지원
7 }
```

- `std::erase()`: 리턴 값으로 제거된 요소의 개수를 반환한다
- 인자로는 컨테이너만 전달 가능하다 (반복자 불가능)

3.3.4 Algorithm with function as parameter

함수를 인자로 가지는 `std::transform`, `std::for_each` 알고리즘에 대해 살펴보자

```
1 int square(int a) { return a*a; }
2 int add(int a, int b) {return a+b; }
3 void print(int n) { std::print("{} , ", n); }

4
5 int main(){
6 std::vector v1{1,2,3,4,5};
7 std::vector<int> v2;
8 std::vector<int> v3;

9
10 std::ranges::transform(v1, std::back_inserter(v2), square); // v2: 1,4,9,16,25 v1을 제곱하여 v2에
11     추가한다
12 std::ranges::transform(v1, v2, std::back_insertter(v3), add); // v3: 2,6,12,20,30 v1과 v2를 더하여 v3에
13     추가한다
14 std::ranges::for_each(v3, print);                                // v3의 요소를 print를 통해 출력한다
15 }
```

- 단항 함수(unary function), 이항 함수(binary function)은 인자가 각각 1,2개인 함수를 말한다
- 각 알고리즘이 단항 함수를 요구하는지 이항 함수를 요구하는지 정확히 알아야 한다.

알고리즘에 전달되는 함수는 "지역 변수를 캡쳐할 수 있는" 장점을 가진다.

```
1 int main(){
2 std::vector v1{1,2,3,4,5};
3 std::vector<int> v2;

4
5 int k = 2;
6 std::ranges::transform(v1,
7 std::back_inserter(v2),
8 [k](int n) { return n + k; } ); // 지역 변수 캡쳐 가능
9 }

10 // int add_k(int n) { return n+k; } // 일반 함수를 사용하는 것은 권장되지 않는다.
```

3.3.5 Predicate

벡터 1,2,6,3,5에서 3의 배수를 찾고자 하는 경우를 보자

```
1 int main(){
2 std::vector v{1,2,6,3,5};

3
4 auto ret1 = std::ranges::find(v, 3);           // 벡터 내에서 3을 찾는다
5 auto ret2 = std::ranges::find_if(v,             // find_if를 사용하면 두번째 파라미터로
6     [=](int n) { return n%3 == 0; } );          // 단항함수(또는 조건자 predicate)를 넣을 수 있다
7 std::println("{} , {}", *ret1, *ret2);
8 }
```

- 조건자(**predicate**)는 bool 또는 bool로 변환 가능한 값을 반환하는 단항 함수를 말한다.
- `std::xxx_if` 형태로 구현된 함수에서 두번째 파라미터에 들어간다.

알고리즘의 조건자 버전 (`xxx_if`)

- `sort` 알고리즘은 `_if`를 사용하지 않는다.
- `find`도 `sort`처럼 조건자 버전의 이름을 동일하게 `find`로 하면 편하지 않았을까?
- c++98 시절의 문법으로는 인자의 갯수가 동일한 함수 템플릿을 같은 이름으로 여러개 만들 수는 없었다
(c++20 기술로는 concept을 사용하여 가능)

```
1 int main(){
2 std::vector v{1,2,6,3,5};
3 }
```

```

4 auto ret1 = std::ranges::find(v.begin(), v.end(), 3);
5 auto ret2 = std::ranges::find_if(v.begin(), v.end(),
6 [](int n) { return n%3 == 0; });
7
8 std::ranges::sort(v.begin(), v.end());
9 std::ranges::sort(v.begin(), v.end(), std::greater{}); // sort는 sort_if처럼 쓰지 않는다
10 }
```

std::predicate concept

- 조건자의 요구사항을 정의한 concept
- std::predicate<T, Type>: 단항 조건자 조사
- std::predicate<T, Type1, Type2>: 이항 조건자 조사

```

1 template<typename T>
2 void is_predicate(T f){
3     std::println("{}", std::predicate<T, int>);
4 }
5
6 int main(){
7     is_predicate([](int n) { return n%3 == 0; }); // true
8     is_predicate([](int a, int b) { return a < b; }); // false. 단항 조건자가 아니므로
9     is_predicate([](double a) { return a; }); // true
10    is_predicate([](int n) { std::println("{}", n); }); // false. 리턴값이 없으므로
11 }
```

3.3.6 Algoritmh copy version

STL의 일부 알고리즘을 복사 버전을 제공한다.

- 알고리즘 이름이 "_copy"로 끝나는 알고리즘
- 연산의 수행 결과를 자신이 아닌 "다른 컨테이너에 저장"하는 알고리즘

```

1 int main(){
2     std::vector v1{1,2,3,1,2,3,1,2,3,1};
3     std::vector v2{0,0,0,0,0,0,0,0,0,0};
4
5     auto ret1 = std::ranges::remove(v1, 3);
6     auto ret2 = std::ranges::remove_copy(v1, v2.begin(), 3); // v1에서 3이 제거된 결과를 v2에 저장한다
7 }
```

- std::remove의 반환값: v1의 반복자(end)를 제공한다
- std::remove_if의 반환값: v3의 반복자(end)를 제공한다

c++20 constrained 알고리즘 복사버전의 반환값은 조금 복잡하다

```

1 int main(){
2     std::vector v1{1,2,3,1,2,3,1,2,3,1};
3     std::vector v2{1,2,3,1,2,3,1,2,3,1};
4     std::vector v3{0,0,0,0,0,0,0,0,0,0};
5
6     auto ret1 = std::ranges::remove(v1.begin(), std::next(v1.begin(),7), 3); // v1에서 처음 7개
7         범위에서 값을 지운다
8
9     auto ret2 = std::ranges::remove_copy(v2.begin(), std::next(v2.begin(),7), v3.begin(), 3); // v2에서
10        처음 7개 범위에서 값을 지우고 v3에 복사
11 }
```

- ret1의 타입은 std::ranges::subranges 타입이다
- ret1.begin(): 7개 범위 내에서 원소 제거 후 end()를 가르키는 반복자 - ret1.end(): 8번째 위치를 가르키는 반복자 (범위가 7이었으므로 그 다음 원소)
- ret2의 타입은 std::ranges::remove_copy_result<I,O> 타입이다 (std::ranges::in_out_result<I,O> 타입의 alias)

-
- ret2.in: v2의 8번째 원소를 가르키는 반복자
 - ret2.out: v3의 end를 가르키는 반복자

STL의 일부 알고리즘은 4가지 변형 버전을 제공한다 (기본형, _if, _copy, _copy_if)

```

1 int main(){
2 std::vector v1{1,3,6,3,5};
3 std::vector v2{0,0,0,0,0};
4
5 auto predicate = [](int n){ return n%3 == 0; }
6
7 // c++98
8 auto ret1 = std::remove(v1.begin(), v1.end(), 3);
9 auto ret2 = std::remove_if(v1.begin(), v1.end(), predicate);
10 auto ret3 = std::remove_copy(v1.begin(), v1.end(), v2.begin(), 3);
11 auto ret4 = std::remove_copy_if(v1.begin(), v1.end(), v2.begin(), predicate);
12
13 // c++20 constrained algorithm
14 auto ret5 = std::ranges::remove(v1, 3);
15 auto ret6 = std::ranges::remove_if(v1, predicate);
16 auto ret7 = std::ranges::remove_copy(v1, v2.begin(), 3);
17 auto ret8 = std::ranges::remove_copy_if(v1, v2.begin(), predicate);
18 }
```

3.3.7 Projection

c++20에서 추가된 projection에 대해 알아보자. Point 객체를 보관하는 컨테이너에서 y가 3인 요소를 찾고 싶다면 어떻게 할까?

```

1 struct Point {
2     int x,y;
3 }
4
5 int main(){
6     std::vector<Point> v{{1,1}, {2,2}, {3,3}, {4,4}};
7
8     auto ret1 = std::ranges::find_if(v,
9         [] (const Point& pt) { return pt.y == 3;}); // 조건자 사용
10    auto ret2 = std::ranges::find(v, 3,&Point::y); // projection 사용 (c++20)
11 }
```

- **Projection:** 컨테이너의 모든 요소를 한 개씩 projection에 통과시켜서 나오는 결과가 3인 것을 검색한다

```

1 struct Point {
2     int x,y;
3     int get_y() const { return y; }
4 }
5
6 int main(){
7     std::vector<Point> v{{1,1}, {2,2}, {3,3}, {4,4}};
8
9 // Projection
10    auto ret1 = std::ranges::find(v, 3, &Point::y); // 멤버 데이터 포인터
11    auto ret2 = std::ranges::find(v, 3, &Point::get_y()); // 멤버 함수 포인터
12    auto ret3 = std::ranges::find(v, 3, // 단항 함수
13        [] (const Point& p){ return p.y; });
14
15 // find_if
16    auto ret3 = std::ranges::find_if(v, 3, // find_if는 단항 조건자만 가능하다
17        [] (const Point& p){ return p.y == 3; });
18 }
```

- projection에 넣을 수 있는 것들은 (1) 멤버 데이터에 대한 포인터, (2) 멤버 함수에 대한 포인터, (3) 단항 함수 (함수 객체)가 들어갈 수 있다.

c++98	<p>‘std’ 이름 공간 인자로 반복자만 사용 가능 함수(템플릿)으로 구현</p>
c++20	<p>‘std::ranges’ 이름 공간 인자로 반복자 뿐만 아니라 컨테이너도 전달 가능 보다 많은 정보를 담은 반환값 함수가 아닌 함수 객체로 구현</p>

- 단항 함수와 단항 조건자(bool return)를 헷갈리지 않도록 유의한다!
- 위 코드는 컨테이너 버전이지만 반복자 버전(v.begin(), v.end())도 가능하다.
- `find` 뿐만 아니라 `find_if`에서도 가능하다

```

1 // find_if + projection
2 auto ret3 = std::ranges::find_if(v,
3 [](int n) { return n == 1; },
4 &Point::y);

```

- `find`, `find_if` 뿐만 아니라 대부분에 알고리즘에서도 projection을 지원한다

Projection을 사용하여 벡터 안에 있는 문자열을 사전 순서가 아닌 문자열의 길이를 기준으로 정렬해보자

```

1 int main(){
2     std::vector<string> v{"AAAA", "D", "BB", "CCC"};
3
4     std::ranges::sort(v);                                // 기본형 (사전 순서)
5
6     std::ranges::sort(v, [](std::string& s1, std::string& s2), // 조건자 사용 (길이 순서)
7     { return s1.size() < s2.size(); });
8
9     std::ranges::sort(v, {}, &std::string::size);          // Projection 사용. 이항 함수는 기본형이 less
10    이므로 를 전달하면 less가 호출됨
11
12    std::ranges::sort(v, std::greater{}, &std::string::size); // Projection 사용. greater 버전
}

```

3.3.8 Constrinaed algorithm function object

c++20에 추가된 constrained algorithm이 함수 객체라는 사실에 대해 자세히 알아보자.

```

1 int main(){
2     auto ret1 = std::max(1, 2);                      // 함수
3     auto ret2 = std::ranges::max(1, 2);                // 함수 객체
4     auto ret3 = std::ranges::max.operator()(1, 2); // 함수 객체
5 }

```

Argument Dependent Lookup (ADL)

- 함수를 찾을 때 인자가 속해 있는 이름 공간은 자동으로 검색하도록 하는 기법

```

1 namespace Graphics {
2     struct Point {
3         int x,y;
4     }
5     void draw_pixel(const Point& pt) {}
6 };
7
8 int main(){
9     Graphics::Point pt{1,2};
10    Graphics::draw_pixel(pt);
11    draw_pixel(pt);           // ok. 이름 공간을 명시하지 않았으므로 안될 것 같으나 ADL에 의해 가능하다
12 }

```

- pt는 Graphics 이름 공간 안에 있으므로 `draw_pixel`도 같은 Graphics 이름 공간 안에 있는지 자동으로 검색한다.

ADL이 필요한 이유: 특정 이름 공간 안에 있는 객체 등에 연산자 재정의(operator overloading) 문법 등을 사용하려면 ADL 문법이 필요하다.

```
1 namespace Graphics {
2     struct Point {
3         int x,y;
4     }
5
6     Point operator+(const Point& p1,
7         const Point& p2) {
8         return Point{p1.x + p2.x, p1.y + p2.y};
9     }
10 };
11
12 int main(){
13     Graphics::Point p1{1,1};
14     Graphics::Point p1{2,2};
15
16     auto ret1 = Graphics::operator+(p1, p2); // 보통 이렇게 작성하지 않는다
17     auto ret2 = p1 + p2; // operator+(p1,p2)가 호출되는데 ADL이 없으면 컴파일 에러가
18     발생한다
19 }
```

```
1 int main(){
2     int n1 = 10;
3     int n2 = 20;
4
5     std::string s1 = "AA";
6     std::string s2 = "BB";
7
8     auto ret1 = std::max(n1, n2); // ok
9     auto ret2 = std::max(n1, n2); // ok
10    auto ret3 = max(n1, n2); // error
11    auto ret4 = max(n1, n2); // ok. string이 std 이름 공간 안에 있으므로 ADL이 적용됨
12 }
```

ADL로 인해 컴파일 에러가 발생하는 예시를 살펴보자

```
1 namespace mystd {
2     class string {};
3
4     template<typename T>
5     void max(const T& a, const T& b) { std::println("std::max"); }
6
7     namespace ranges {
8         template<typename T>
9         void max(const T& a, const T& b) { std::println("std::ranges::max"); }
10    }
11 }
12
13 int main(){
14     mystd::string s1, s2;
15
16     mystd::max(s1, s2); // ok
17     mystd::ranges::max(s1, s2); // ok
18
19     using namespace mystd::ranges; // 신버전 이름 공간을 열어놓은 경우를 살펴보자
20     max(s1, s2); // error! ADL 이름 공간과 열어놓은 mystd::ranges 이름 공간이 같은 이름의
21     함수로 충돌한다
22 }
```

- 위와 같은 ADL 이름 충돌을 막기 위해 c++20에서는 constrained algorithm들을 **함수가 아닌 함수 객체로 구현**하였다.

```
1 namespace mystd {
2     class string {};
3
4     template<typename T>
5     void max(const T& a, const T& b) { std::println("std::max"); }
6
7     namespace ranges {
8         struct max_fn {
9             template<typename T>
10            void operator()(const T& a, const T& b) { std::println("std::ranges::max"); }
11        };
12        max_fn max;           // 함수 객체로 구현되었다
13    }
14 }
```

4 References

- [1] (lecture) CODENURI - C++ Master
- [2] (blog) [모던C++] 정리 - tango1202

5 Revision log

- 1st: 2024-07-16
- 2nd: 2024-07-23
- 3rd: 2024-07-30
- 4th: 2024-08-01
- 5th: 2024-08-11
- 6th: 2024-12-13